UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Sistemas Informáticos y Programación

UNA APROXIMACIÓN ONTOLÓGICA AL DESARROLLO DE SISTEMAS DE RAZONAMIENTO BASADO EN CASOS

MEMORIA PARA OPTAR AL GRADO DE DOCTOR
PRESENTADA POR

María Belén Díaz Agudo

Bajo la dirección del Doctor:

Pedro Antonio González Calero

Madrid, 2002

ISBN: 84-669-1851-5

UNA APROXIMACIÓN ONTOLÓGICA AL DESARROLLO DE SISTEMAS DE RAZONAMIENTO BASADO EN CASOS

Memoria que presenta para optar al grado de
Doctora en Informática

Mª Belén Díaz Agudo

Dirigida por el profesor
Pedro A. González Calero

Departamento de Sistemas Informáticos y Programación
Facultad de Informática
Universidad Complutense de Madrid

Septiembre 2002

Agradecimientos

Parece mentira que después de tanto tiempo y tanto esfuerzo por fin esté escribiendo estos agradecimientos. En todos estos años son muchas las personas que me han ayudado y apoyado y sin ellas nunca habría terminado esta tesis.

En primer lugar, quiero dar las gracias a Pedro González mi director de tesis durante estos años y un buen amigo. Gracias por dejarme trabajar contigo y por enseñarme tu *método* de trabajo. Gracias por compartir tu tiempo, por tus ideas, por enriquecer mi trabajo con tus comentarios y por tu gran capacidad de síntesis que tanto me ha ayudado. Gracias por ser mi compañero de congresos, por soportarme cuando fui insoportable y por tu paciencia. Sobre todo gracias por tu gran paciencia en esos momentos malos que a veces tengo y que los que me conocen tienen el *privilegio* de sufrir. Gracias también por compartir los momentos buenos. Gracias por entenderme, por escucharme y por animarme en los momentos bajos. Gracias por todas las visitas al hospital (gracias a ti también, Eva).

Y ya que ha salido el tema quiero agradecer de corazón el soporte, el cariño y las numerosísimas llamadas y visitas que recibí en esa fatídica etapa de mi vida. Gracias a Dios por permitirme olvidarla y por enseñarme a apreciar las cosas que son realmente importantes en la vida. Gracias por dejarme estar donde estoy.

Volvemos a los buenos momentos, y con ellos a los que han sido mis compañeros de trabajo durante los últimos años. Cronológicamente tengo que empezar por Carmen Fernández, ya que ella fue mi punto de contacto con el departamento hace unos años y, además de la Inteligencia Artificial, me explicó los pasos que debía dar para ser profesora de universidad, incluyendo la tesis, y aquí estoy. Gracias por tus buenos consejos en ese y otros aspectos. A Mercedes Gómez gracias por las exhaustivas revisiones de los capítulos de esta memoria que han mejorado en gran medida la coherencia y claridad de este trabajo. Gracias por dedicarme tu tiempo y por animarme cuando lo necesité. A Pablo Gervás le agradezco todo lo que me ha enseñado sobre el apasionante mundo de la poesía. Gracias por compartir tu trabajo conmigo. A mi amigo (y ex-compañero de despacho) Carlos Fernández quiero agradecerle el soporte técnico que nos dio a todos. Gracias por animarnos los viernes por la tarde. Esto no es lo mismo sin ti.

A mis otros compañeros de despacho, y en especial a Eva Ullán, a Natalia López y a mi compañero actual, Jaime Sánchez, quiero agradecerles el buen ambiente de trabajo gracias al que he conseguido terminar esta tesis. Eva fue mi primera compañera y la que me ayudó a *aterrizar* en el departamento. Gracias por aquellas largas conversaciones con las que nos hicimos amigas. A Natalia le agradezco que compartiera mis nervios y que supiera tranquilizarme. Y a Jaime le agradezco su forma de ser, esa filosofía que tiene de la vida que lo hace parecer todo tan fácil.

Gracias al resto de mis compañeros del Departamento de Sistemas Informáticos y Programación y a mis compañeros del grupo ISIA por proporcionarme un entorno de trabajo agradable. En especial, además de los ya citados, a Antonio Vaquero, Alfredo Fernández, Baltasar Fernández, Antonio Navarro, José Luis Sierra (y sus mónadas), Héctor Gómez, Jorge Gómez, Juan Pavón y Rafael Caballero. Gracias a Luis Hernández por solucionarme todos mis asuntos, dudas y trámites con amabilidad y eficacia (y sin llamarme pesada aunque me lo mereciera). Gracias a Bonifacio de Andrés por un buen consejo que me dio una vez.

Gracias a todos los que fueron mis profesores en la Escuela Superior de Informática (ahora Facultad) y que ahora son compañeros del departamento. Gracias por todo lo que aprendí.

También quiero dar las gracias a todos mis amigos que tantas veces me han preguntado por el estado de la tesis y me han animado. En especial a Idoia, Patricia y Sagri que tan orgullosas están de mí. Ellas ha sido mis mejores amigas de la facultad y compañeras de estudio, de dudas, de prácticas, de alegrías y de penas. Gracias también al resto de mis compañeros de la facultad por compartir conmigo sus *mejores* años.

Gracias a todos aquellos que estén leyendo estos agradecimientos como preludio de la lectura de esta tesis. Gracias por tomaros interés en mi trabajo.

Casi para terminar quiero dar las gracias a familia, en especial a mis hermanos, Antonio, Begoña y Esther, y a mis otros hermanos: Alberto, Angélica, Cristina, Pablo, Paloma, Roberto, Ana, Paul, Patricia, Ian, Macarena, Giovanni y Alex, por quererme como soy. Gracias a Laura, Oliver, Sara y Lara (casi) por alegrarme la vida. Gracias Laura por esperarme. Gracias a Dori e Ignacio por las vacaciones.

He dejado para el final a alguien que se merece mi gratitud eterna. Gracias Juanjo por todos estos años. Nunca podré agradecerte lo bastante tu apoyo incondicional, tus consejos y tu amor. Aunque mucha gente me ha acompañado en este camino, sin ti nunca hubiera llegado al final. Gracias por animarme en esos momentos que sólo tu has sufrido y por compartir mis angustias y mis agobios. Gracias por ayudarme a salir del *bache*. Espero de verdad poder compensarte algún día y ayudarte en tu camino tanto como me has ayudado tú en el mío. Te quiero.

Por último, quiero agradecer a mis padres, su amor, su apoyo y su esfuerzo para que consiguiera mis objetivos. Gracias por enseñarme que el trabajo siempre tiene recompensa, muchas veces me acordé y aquí estoy. Gracias por todo.

Índice

Capítulo 1
INTRODUCCIÓN

Nuestro trabajo en esta tesis se ha centrado en el diseño de sistemas de razonamiento basado en casos (en inglés *Case-Based Reasoning*, CBR) que integren procesos de razonamiento con conocimiento adicional sobre el dominio de aplicación. En concreto hemos estudiado las ventajas de disponer de un modelo terminológico de conocimiento del dominio explícitamente representado, así como también los beneficios derivados de utilizar las lógicas descriptivas (en inglés *Description Logics*, DLs): un formalismo de representación de conocimiento que proporciona mecanismos de razonamiento sofisticados.

Los sistemas CBR razonan con conocimiento específico de situaciones concretas previamente experimentadas (casos). Algunos sistemas completan el conocimiento de estos casos con un depósito de conocimiento general acerca de un dominio, permitiendo procesos de razonamiento más completos, pero a la vez incrementando el coste de adquisición del conocimiento involucrado en el sistema. Una idea subyacente a esta tesis consiste en promover la reutilización de componentes de conocimiento durante el diseño de sistemas CBR. En concreto nos interesa aplicar al diseño de sistemas CBR los resultados de dos áreas de investigación activas en Inteligencia Artificial (IA): las *ontologías* y los *métodos de resolución de problemas*. La reutilización de ontologías como fuente de conocimiento sobre CBR y sobre el dominio de aplicación, teniendo en cuenta que una ontología define los términos y relaciones básicos que comprenden el vocabulario de un área de aplicación. Por otro lado, los métodos de resolución de problemas permiten representar el conocimiento asociado con los procesos CBR independientes del dominio que se pueden reutilizar en otros sistemas.

Las aportaciones fundamentales de nuestro trabajo son dos: (i) la representación explícita -utilizando un sistema de DLs- de CBROnto, una ontología sobre la resolución de problemas mediante CBR, tanto respecto a la terminología -vocabulario- como a las tareas y métodos típicamente asociados a los sistemas CBR, y (ii) el diseño e implementación de COLIBRI (*Cases and Ontology Libraries Integration for Building Reasoning Infrastructures*), un entorno de desarrollo de sistemas CBR basado en CBROnto.

1. Motivación y Objetivos

Cualquier sistema basado en conocimiento (en inglés *Knowledge Based System*, KBS) se basa en la representación explícita de una gran cantidad de conocimiento, organizado de manera efectiva, con el que el sistema es capaz de razonar. El conocimiento puede ser de distinta naturaleza, aunque los sistemas "tradicionales" se construyen alrededor de un *modelo de comportamiento del dominio* que consiste, básicamente, en un conjunto de hechos y un conjunto de reglas que permiten deducir nuevos hechos aplicando un *motor de inferencia*.

Los problemas que históricamente se asocian al desarrollo de KBSs se refieren básicamente al llamado *cuello de botella de la adquisición de conocimiento*. En el desarrollo de un nuevo sistema, la adquisición de conocimiento se convierte, muchas veces, en un problema insalvable. Para intentar solventar el problema, las principales metodologías de análisis y desarrollo de KBSs coinciden en aspectos fundamentales relativos a la conceptualización y representación de componentes de conocimiento reutilizables –ontologías, estructuras de tareas y métodos de resolución de problemas– y en la reutilización de dichos componentes como la base fundamental del desarrollo de nuevos KBSs.

En otra línea de trabajo, el razonamiento basado en casos representa un enfoque de resolución de problemas y aprendizaje dentro de la IA que surge como una posible solución al problema de la adquisición de conocimiento asociado con los sistemas "tradicionales". El CBR es un conjunto de técnicas para el desarrollo de KBSs que se basa en almacenar, recuperar y reutilizar las soluciones a problemas parecidos previamente resueltos, en lugar de generar soluciones basadas en un modelo general exhaustivo de conocimiento del dominio. El CBR facilita la adquisición de conocimiento porque a los expertos les resulta más sencillo recordar hechos pasados que proporcionar reglas de aplicación general. Aunque el CBR no elimina el problema de la adquisición de conocimiento, sí lo alivia, si aceptamos que los modelos necesarios son más "ligeros" que los de un sistema basado en reglas equivalente.

Aunque cualquier sistema de CBR basa su razonamiento en el conjunto de experiencias previas, algunas aproximaciones al CBR complementan el conocimiento específico de los casos con otros tipos de conocimiento *general* sobre el dominio. Estas aproximaciones están justificadas por el hecho de que un sistema podrá ser más efectivo si dispone de más conocimiento con el que razonar, pero requieren un equilibrio para no derivar en un nuevo cuello de botella. Aunque este conocimiento general puede ser de distintos tipos, nuestra aproximación se basa en sistemas CBR complementados con modelos terminológicos sobre el dominio y que llamaremos sistemas KI-CBR (del inglés, *Knowledge Intensive* CBR).

El principal objetivo del trabajo que presentamos es facilitar el proceso de diseño de sistemas KI-CBR. Para ello proponemos un esquema de adquisición de conocimiento basado en la reutilización de ontologías del dominio y una arquitectura basada en CBROnto, una ontología sobre CBR que hemos desarrollado y que proporciona terminología y una biblioteca de métodos reutilizables que se configurarán durante el diseño de nuevos sistemas KI-CBR. Esta arquitectura se plasma en COLIBRI, un sistema de ayuda para el diseño de sistemas KI-CBR.

Otro aspecto fundamental de nuestro trabajo se refiere a la formalización del conocimiento que interviene en los KBSs. La investigación en IA ha producido numerosos formalismos para la representación de conocimiento con ciertos mecanismos de razonamiento asociados. En este trabajo, hemos estudiado las ventajas que plantea el uso de las *lógicas descriptivas* como tecnología de representación de conocimiento. Nuestra elección está justificada por un conjunto de características –principalmente la capacidad de razonamiento con el conocimiento

representado- que han convertido a las lógicas descriptivas en una tecnología muy adecuada para la formalización de ontologías y para la formalización de todo el conocimiento involucrado en un sistema CBR.

Con el espíritu de promover la reutilización de conocimiento a través del uso de bibliotecas de componentes, COLIBRI propone diseñar aplicaciones KI-CBR utilizando el siguiente esquema:

1. Construir un modelo terminológico de conocimiento general sobre el dominio. Para aliviar el coste de adquisición de este conocimiento se propone la reutilización de ontologías del dominio previamente formalizadas. Una ontología del dominio proporciona un conjunto de términos jerárquicamente estructurados para describir la terminología de ese dominio.
2. Utilizar la terminología del dominio, adquirida en el paso anterior, junto con la terminología de CBROnto para definir los casos de la biblioteca inicial.
3. En función de las características de la aplicación y del dominio, elegir los métodos CBR independientes del dominio que se incluyen en la biblioteca de métodos de CBROnto.
4. Para que estos métodos sean aplicables en un dominio concreto será necesario un paso de integración que relacione la terminología de CBROnto usada en la definición de los métodos genéricos con la terminología del dominio concreto. El mecanismo que utilizamos para llevar a cabo la integración de conocimiento es la clasificación semántica de las lógicas descriptivas. La terminología de CBROnto proporciona la infraestructura que sirve como "pegamento" sintáctico y semántico entre la terminología del dominio y los métodos CBR genéricos que se reutilizan.

A continuación se relacionan los principales objetivos de esta tesis:

- Conceptualizar CBROnto, una ontología con conocimiento sobre CBR que incluye:
    - Terminología general sobre CBR incluyendo un lenguaje de representación de casos.
    - Una biblioteca de métodos de resolución de problemas, definidos con dicha terminología, que resuelven las tareas CBR y que pueden ser reutilizados en sistemas CBR en distintos dominios.
- Formalizar CBROnto utilizando LOOM, un sistema de lógica descriptiva.
- Proponer una metodología de diseño de sistemas CBR basada en la reutilización, por un lado, de terminología sobre el dominio de una biblioteca de ontologías y, por otro lado, de terminología sobre CBR de CBROnto.
- Estudiar si el tipo de conocimiento que encontramos en las ontologías del dominio es o no adecuado para llevar a cabo procesos CBR sofisticados que usan activamente este conocimiento además de los casos.
- Estudiar la adecuación de las lógicas descriptivas como tecnología de representación del conocimiento para representar y organizar todo el conocimiento necesario por nuestro esquema. Estudiar la utilidad de los mecanismos de razonamiento de las lógicas descriptivas para:
    - Adquirir conocimiento del dominio a través de la reutilización de ontologías del dominio de una biblioteca.
    - Integrar el conocimiento del dominio con el conocimiento sobre CBR de CBROnto.

- Desarrollar COLIBRI, un entorno de desarrollo que permita el diseño y prototipado de aplicaciones KI-CBR. La arquitectura de COLIBRI se basa en las ideas anteriores y en el uso de CBROnto como núcleo básico.

2. Estructura de la tesis

La memoria de esta tesis está organizada en siete capítulos, incluyendo este primero de introducción, y cinco apéndices.

En el Capítulo 2 revisamos los conceptos fundamentales de los sistemas CBR, en general, y de los sistemas KI-CBR, en particular, haciendo énfasis en el conocimiento, tanto el que está representado explícitamente en los casos, como aquel que se encuentra involucrado en los distintos procesos CBR. Introducimos los conceptos básicos y las líneas de investigación actuales de la comunidad CBR, y repasamos algunas de las aproximaciones y técnicas clásicas utilizadas.

En el Capítulo 3 describimos algunas aproximaciones que proponen compartir y reutilizar conocimiento para construir KBSs. En particular, se definen las características de dos tipos de componentes reutilizables que intervienen en las principales metodologías de construcción de KBSs: las ontologías y los métodos de resolución de problemas. Aunque para formalizar dichos componentes se podrían utilizar diversas técnicas de representación de conocimiento, en este capítulo describimos las ideas básicas de la opción elegida en nuestro trabajo: las *lógicas descriptivas*. Una de las propuestas de esta tesis consiste en promover la reutilización del conocimiento proporcionando y utilizando bibliotecas de componentes para el diseño de aplicaciones KI-CBR. Los conceptos presentados en este capítulo justifican las ventajas que hacen de la reutilización de conocimiento una aproximación adecuada, además de para los KBSs en general, para los sistemas KI-CBR en particular. Adicionalmente justificamos los aspectos que hacen de las lógicas descriptivas un formalismo idóneo para la representación de conocimiento ontológico y para el desarrollo de sistemas KI-CBR.

El núcleo de esta memoria se encuentra en los Capítulos 4 y 5 donde se describe CBROnto, una ontología que captura términos semánticamente importantes para los sistemas CBR y que está formalizada en un sistema de lógica descriptiva. Las definiciones de CBROnto se refieren a CBR en general independientemente del dominio de aplicación. Además de la terminología sobre CBR, CBROnto incluye conocimiento sobre tareas y métodos CBR, es decir, representaciones explícitas de métodos que resuelven las tareas involucradas en el CBR y que pueden ser reutilizados en distintos dominios.

El Capítulo 4 introduce las ideas fundamentales y la organización de la terminología, tareas y métodos de CBROnto. En relación con la organización de los casos describimos una técnica inductiva que hemos utilizado en nuestro trabajo para extraer conocimiento de una base de casos: el *Análisis Formal de Conceptos*.

El Capítulo 5 detalla cada uno de los métodos de la biblioteca de CBROnto. Para cada método se describe su comportamiento incidiendo fundamentalmente en cómo aprovecha los mecanismos de razonamiento de las DLs como pasos básicos de inferencia.

En el Capítulo 6 describimos el sistema COLIBRI, cuyo objetivo es ayudar a diseñar aplicaciones KI-CBR reutilizando componentes, principalmente ontologías del dominio y CBROnto como ontología que proporciona terminología y métodos reutilizables que resuelven las tareas CBR. El diseño de aplicaciones KI-CBR utilizando COLIBRI se ejemplifica en este capítulo con el diseño de una aplicación CBR para generar poesía en castellano.

Por último en el Capítulo 7 resumimos las conclusiones principales del trabajo desarrollado en esta tesis y proponemos algunas líneas de trabajo futuro.

Después de reseñar la bibliografía utilizada en el desarrollo de este trabajo de tesis la memoria finaliza con los siguientes apéndices:

- Apéndice A: resumen de la sintaxis de LOOM, el sistema de representación de conocimiento basado en lógicas descriptivas que hemos utilizado para formalizar CBROnto.
- Apéndice B: formalización de CBROnto en LOOM.
- Apéndice C: listado (texto informal) de la biblioteca de métodos de CBROnto.
- Apéndice D: listado de funciones de la API de COLIBRI.
- Apéndice E: formalización en LOOM del modelo del dominio de la aplicación de generación de poesía.

Capítulo 2
EL RAZONAMIENTO BASADO EN CASOS

1. Introducción

El razonamiento basado en casos (CBR) es un enfoque de resolución de problemas y aprendizaje dentro del área de la Inteligencia Artificial (IA) relativamente reciente, ya que podemos situar sus raíces a finales de los setenta.

Entendido como metodología de construcción de sistemas basados en conocimiento (KBSs) el CBR es diferente a otros enfoques de la IA. Así, en lugar de basarse en reglas que modelen el comportamiento del dominio del problema o en la generalización de relaciones entre descriptores de problemas y conclusiones, el CBR utiliza el conocimiento específico obtenido a partir de situaciones concretas previamente experimentadas: los casos. Una nueva situación se afronta localizando un caso previo similar y adaptando la solución antes obtenida para que se adecue a la situación actual. Otra diferencia importante del CBR respecto a otros enfoques es el aprendizaje incremental: el sistema actualiza su conocimiento en base a nuevas experiencias que pasan así a estar disponibles para abordar futuros problemas.

Cualquier KBS se basa en la representación explícita de una gran cantidad de conocimiento, organizado de manera efectiva, con el que el sistema es capaz de razonar. El CBR surge como una posible solución al problema fundamental que ha frenado durante años el desarrollo de KBSs: el llamado *cuello de botella de la adquisición de conocimiento*. Un KBS se construye alrededor de un *modelo del dominio* que, mediante los mecanismos de razonamiento adecuados, permite simular el comportamiento de un experto del dominio. La investigación en IA ha producido numerosos formalismos para la representación de conocimiento así como potentes mecanismos de razonamiento asociados, sin embargo, cuando se intentan aplicar estos resultados a gran escala, la adquisición de conocimiento se convierte, muchas veces, en un problema insalvable: ¿cómo conseguir representar explícitamente todo lo que un experto –o una comunidad de expertos– sabe? El CBR facilita la adquisición de conocimiento porque para empezar a funcionar sólo necesita un conjunto de problemas resueltos.

Comenzamos este capítulo con una descripción de los KBSs que sirve de contexto a la introducción sobre los orígenes e ideas principales del CBR que se presenta en el Apartado 3. El Apartado 4 se encarga de los fundamentos del CBR y de los procesos involucrados haciendo hincapié en el conocimiento y los mecanismos de razonamiento que se utilizan para resolverlos, y teniendo en cuenta que el conocimiento almacenado en los casos no es obligatoriamente el único con el que cuenta un sistema CBR. Por último el Apartado 5 repasa algunas herramientas y aplicaciones CBR.

2. Sistemas basados en conocimiento

El origen de los KBSs data de principios de la década de los 70, cuando algunos investigadores observaron que los métodos y técnicas de búsqueda generales desarrollados durante la década anterior, resultaban insuficientes para resolver problemas con cierto grado de dificultad. En lugar de la tendencia hasta ese momento, de manejar conocimiento de naturaleza genérica, es decir aplicable a varios dominios, convinieron en la alternativa de disponer de conocimiento específico sobre el dominio de aplicación particular que fuese de interés en cada aplicación concreta. Desde sus comienzos, los KBSs han sido un área fundamental de la IA con un gran impacto tanto en el ámbito de investigación como en la industria.

En el marco de las Tecnologías de la Información, es habitual distinguir entre Sistemas Convencionales y KBSs como productos respectivos de dos Ingenierías distintas: la del Software y la del Conocimiento. Los problemas objeto de la Ingeniería del Software son sistemáticos, procedimentales y con resolución algorítmica, mientras que la Ingeniería del Conocimiento aborda problemas de resolución eminentemente heurística. Dicho de otro modo, los sistemas convencionales suelen aplicarse en dominios relacionados con el tratamiento eficaz de los datos e involucran un conjunto de instrucciones que generan una solución única y correcta, mientras que los basados en conocimiento abordan problemas que requieren *inteligencia* y conocimiento especializado, para los que no existe una solución algorítmica o ésta es demasiado compleja o ineficiente.

El comportamiento inteligente de un KBS se obtiene mediante un *motor de inferencias* que manipula cierto conocimiento representado en una *base de conocimiento* para solucionar un problema dado [González&Dankel97]. El motor de inferencias contiene conocimiento general de resolución de problemas mientras que la base de conocimiento contiene conocimiento específico sobre el dominio del problema. Precisamente el poder de los KBS proviene de esta separación del conocimiento respecto a cómo se utiliza, lo que permite desarrollar aplicaciones en las que la técnica de razonamiento genérico no se modifica, es decir, únicamente se debe codificar la base de conocimiento del dominio de aplicación.

La misión principal de la Ingeniería del Conocimiento es la de adquirir, conceptualizar, formalizar y utilizar conocimiento específico para la resolución de una tarea. En [Gómez-Pérez *et al.* 97] se define la Ingeniería del Conocimiento como:

> *"El conjunto de principios, métodos y herramientas que permiten aplicar el saber científico y de experiencia a la utilización del conocimiento y de sus fuentes, mediante construcciones útiles para el hombre. Es decir, la Ingeniería del Conocimiento encara el problema de construir sistemas computacionales con pericia, aspirando primero a adquirir los conocimientos de distintas fuentes y, en particular, a educir los conocimientos de los expertos y luego a organizarlos en una implementación efectiva"*

El término KBS se ha utilizado frecuentemente como sinónimo del término Sistema Experto, por ejemplo en [González&Dankel97]. En general un KBS es cualquier sistema que se

basa en la representación explícita y el uso de conocimiento sobre un cierto dominio de aplicación. Existen tres conceptos fundamentales que caracterizan a los KBSs:

- La separación del conocimiento de cómo éste es utilizado.
- El uso de conocimiento del dominio.
- La naturaleza heurística frente a la algorítmica del conocimiento empleado.

Existen otros trabajos, por ejemplo [Gómez-Pérez *et al.* 97], que consideran los Sistemas Expertos como un caso particular de KBSs en los que el conocimiento representa la experiencia de un experto humano. Este conocimiento se integra dentro de un sistema computacional capaz de resolver problemas complejos que normalmente requieren un alto nivel de experiencia humana.

Relacionados con los distintos tipos de expertos existen varios tipos de experiencia o conocimiento. Por ejemplo el conocimiento llamado asociativo, consiste en relaciones causa-efecto que se originan de las experiencias pasadas del experto en cuestión y que generalmente se llaman *heurísticas*. Una heurística es cualquier estrategia o truco utilizado para mejorar la eficiencia de un sistema. Las heurísticas presentan un margen de error ya que no representan un análisis exhaustivo del problema, aunque representan una elección aceptable cuando el número de posibilidades a examinar es muy grande o involucran una función algorítmica muy compleja o desconocida. Otro tipo de expertos, más científicos, poseen un conocimiento más teórico y profundo del dominio, lo que les permite resolver problemas que nunca han resuelto previamente. Este tipo de conocimiento es difícil de duplicar en un KBS, aunque los llamados sistemas de razonamiento basado en modelos (*model-based systems*) encapsulan este tipo de conocimiento complejo y razonan con él.

El siguiente apartado describe algunas características asociadas a la adquisición de conocimiento específico del dominio, extraído por el ingeniero del conocimiento de las distintas fuentes a su alcance, y a su representación explícita en una base de conocimiento.

2.1 Representación y adquisición de conocimiento

Como hemos descrito en el apartado anterior, el núcleo de los KBSs es el conocimiento del dominio representado explícitamente con el que el sistema puede razonar. Sin embargo, para obtener un comportamiento inteligente no basta con la representación explícita del conocimiento por sí sola, es necesario además organizarlo de manera efectiva, y razonar de forma que recuperemos el conocimiento adecuado en el momento preciso. Es decir, el éxito de una aproximación al razonamiento basada en conocimiento depende tanto de los procesos que manejan el conocimiento como del conocimiento en sí mismo.

Los KBSs utilizan variaciones de los esquemas clásicos de representación de conocimiento: lógica, reglas, redes asociativas o semánticas, marcos y objetos. El núcleo del proceso de desarrollo de los KBSs, y en concreto de los Sistemas Expertos, es la transferencia y transformación de la experiencia en resolver problemas de una fuente de conocimiento -típicamente un experto- a un programa.

En la comunidad de Sistemas Expertos se ha definido la figura del *ingeniero del conocimiento* como el responsable de "interrogar" a los expertos para obtener el conocimiento con el que razonará el sistema. Idealmente, un ingeniero del conocimiento mantendrá una serie de entrevistas con uno o más expertos donde obtendrá la información necesaria para desarrollar un modelo conceptual que luego explicitará en algún marco de representación. Sin embargo, en la práctica, este proceso está lleno de interrogantes: ¿Hay algún experto dispuesto a dedicar el tiempo necesario para sacar a la luz el conocimiento?¿El experto y el ingeniero del

conocimiento se entienden?¿El ingeniero del conocimiento es capaz de captar todas las sutilezas del dominio para poder así representarlas?¿Es posible formalizar el conocimiento?

Las metodologías de desarrollo de KBSs a menudo contemplan una etapa extensa e importante que se encarga de la adquisición de conocimiento. En concreto, no resulta un proceso sencillo trasladar el conocimiento de un experto humano a una representación abstracta o conceptualización efectiva, así como tampoco la representación del conocimiento en términos de estructuras de información procesables automáticamente. Según la naturaleza y tipo del conocimiento involucrado, para sistemas que no requieran modelos exhaustivos de comportamiento, algunas metodologías de diseño de KBSs suavizan el coste de adquisición de conocimiento basándose en la reutilización de componentes. Por su relevancia en el contexto de esta tesis estos aspectos se estudian en detalle en el Capítulo 3 de esta memoria.

El objetivo de este capítulo, cubierto por los apartados siguientes, es la descripción del enfoque de resolución de problemas de los sistemas CBR que surge como otra posible solución a los problemas de la adquisición de conocimiento que hemos descrito. Además, resuelve un problema más profundo que se plantea cuando no existen los principios generales aceptados por todos los expertos a partir de los cuales construir el modelo, es decir, no existe un modelo, ni siquiera en la mente de los expertos, ya que ni siquiera ellos comprenden hasta el final todas las implicaciones de su actividad diaria.

El CBR facilita la adquisición de conocimiento, en primer lugar, porque a los expertos les resulta más sencillo "contar batallitas" que proporcionar reglas de aplicación general. En principio, un sistema CBR, para empezar a funcionar, sólo necesita un conjunto de problemas resueltos que, serán proporcionados por expertos o incluso pueden estar disponibles porque en el dominio en cuestión se realice algún tipo de registro –por ejemplo un archivo de casos clínicos. En este contexto, el experto se limitaría a proporcionar el conocimiento de similitud entre problemas junto con el conocimiento necesario para adaptar la solución de un problema a otro parecido. El CBR alivia el problema de la adquisición de conocimiento ya que los modelos de conocimiento utilizados son más "ligeros" que los de un sistema equivalente que se base en un modelo exhaustivo de comportamiento.

3. El Razonamiento Basado en Casos

El CBR es una aproximación al desarrollo de KBSs que se basa en almacenar, recuperar y reutilizar las soluciones a problemas parecidos previamente resueltos, en lugar de generar soluciones basadas en un modelo exhaustivo de comportamiento. Las raíces del CBR hay que buscarlas en ciertos resultados de Psicología donde se demuestra que, en muchas ocasiones, los seres humanos resolvemos problemas en base a nuestras experiencias pasadas y no a partir de un conocimiento profundo del problema en cuestión. Los médicos, por ejemplo, buscan conjuntos de síntomas conocidos, los ingenieros toman muchas de sus ideas de soluciones previas ya construidas con éxito, o los programadores expertos reutilizan esquemas más o menos abstractos de las soluciones que conocen.

Además de ser una técnica psicológicamente plausible para modelar el razonamiento humano, desde un punto de vista más técnico el CBR representa una posible solución al problema de la adquisición de conocimiento subyacente a los KBSs tradicionales.

Un sistema CBR necesita una colección de experiencias, llamadas *casos*, almacenadas en una *base de casos*, donde cada caso se compone generalmente de una descripción del problema y la solución que se aplicó. Las hipótesis fundamentales en las que se basa el CBR son, primero, que un sistema –o un ser humano– puede ser un resolutor de problemas eficiente y

efectivo sin necesidad de poseer un conocimiento completo, hasta las últimas consecuencias, de la relación que existe entre un problema y su solución, siempre y cuando tenga suficiente experiencia. Y, segundo, que los problemas tienden a repetirse y, por ello, la experiencia es un recurso útil.

Otro aspecto favorable del CBR como tecnología para el desarrollo de KBSs es que facilita el aprendizaje. En su expresión más sencilla, un sistema CBR puede aprender por simple acumulación de casos; esto es, cada nuevo problema que se plantea al sistema, una vez resuelto, puede incorporarse directamente como un nuevo caso. Parece claro que resulta más sencillo recordar los problemas que se han ido resolviendo que generar nuevas reglas que enriquezcan el modelo de comportamiento.

Otras ventajas del CBR son que permite construir razonadores más eficientes, puesto que suele ser menos costoso modificar una solución previa que construir una nueva solución desde cero; los casos pueden proporcionar también "información negativa", alertando sobre posibles fallos; se facilita el mantenimiento de la base de conocimiento, ya que los usuarios pueden añadir nuevos casos sin ayuda de los expertos; y resulta más sencillo conseguir que los usuarios acepten las sugerencias del sistema que están avaladas por una situación previa.

En resumen, el CBR se considera especialmente adecuado en dominios poco formalizados y donde el aprendizaje juega un papel importante. Si es posible construir fácilmente un modelo formal de comportamiento de un determinado dominio entonces el CBR pierde sentido, ya que el modelo general se supone que permite resolver cualquier problema, mientras que un sistema CBR sólo proporciona variaciones de problemas ya resueltos y almacenados en la base de casos. Por el contrario, si ese modelo no existe, o es demasiado costoso de obtener, el CBR puede ser una aproximación efectiva para resolver problemas típicos, facilitando además la adquisición de conocimiento en forma de nuevos casos. Evidentemente los dos enfoques no son excluyentes, y son muchos los dominios donde un modelo general de comportamiento –que resulte incompleto- se puede complementar con conocimiento concreto en forma de casos.

Janet Kolodner [Kolodner93] describe una serie de características que permiten identificar los dominios en los que una aproximación basada en casos tiene probabilidades de éxito:

- A los expertos no les resulta sencillo dar reglas de comportamiento aunque sí proporcionar ejemplos. Los expertos saben a qué se refieren cuando hablan de "casos", suelen comparar un problema nuevo con casos pasados y resuelven problemas adaptando las soluciones de casos pasados.
- Habitualmente en el proceso de formación se utilizan casos en las explicaciones; hay casos disponibles en la bibliografía y en la experiencia de los expertos.
- Se pueden obtener nuevos casos de la resolución de los nuevos problemas y hay alguna forma de clasificar el resultado de un caso como éxito o fracaso.
- Es posible comparar los casos y adaptarlos de manera efectiva. Los casos se pueden generalizar en alguna medida y es posible abstraer sus características relevantes.
- Los casos mantienen su vigencia durante bastante tiempo, es decir, los problemas tienden a repetirse.

3.1 Origen y evolución

Los primeros trabajos en CBR se deben a Schank y Abelson a finales de la década de los 70 [Schank&Abelson77]. Estos trabajos se basaban en la suposición de que las personas almacenamos el conocimiento en forma de *guiones* que nos permiten hacer predicciones y realizar

inferencias. Estos guiones describían situaciones típicas como ir al médico a comer en un restaurante. Aunque los guiones resultaron ser un modelo incompleto de memoria, sirvieron como base para los trabajos que Roger Schank y su grupo desarrollaron en la Universidad de Yale a principios de los 80.

Fue en el grupo de Yale donde se sentaron las bases del CBR y se desarrollaron las primeras aplicaciones según este modelo. CYRUS, desarrollado por Janet Kolodner [Kolodner83], fue el primer sistema CBR, donde los casos representaban conocimiento acerca de los viajes y las entrevistas del Secretario de estado americano Cyrus Vance. Posteriormente, también en Yale, se desarrollaron MEDIATOR [Simpson85] –un sistema de arbitraje–, CHEF [Hammond89]– para el diseño de recetas de comida china–, PERSUADER [Sycara88] –para la construcción de argumentaciones–, CASEY [Koton89] –para el diagnóstico de enfermedades cardiovasculares– y JULIA [Hinrichs92] –para el diseño de menús–.

Ya en la segunda mitad de los 80, cabe reseñar el trabajo de Bruce Porter, en la Universidad de Texas que dio lugar al sistema PROTOS [Bareiss88] [Bareiss *et al.* 90] aplicado al diagnóstico de enfermedades del aparato auditivo. Además. no es de extrañar que el CBR se aplicara rápidamente al dominio del sistema judicial americano que está muy orientado a la jurisprudencia y donde las argumentaciones legales se construyen a partir de casos pasados. En este ámbito es de destacar el trabajo de Edwina Rissland y Kevin Ashley en el desarrollo del sistema HYPO [Rissland83]. Este sistema, construía argumentos legales a favor y en contra de los litigantes en base a las similitudes y diferencias con otros casos. Posteriormente, el sistema HYPO ha sido aplicado a la enseñanza del derecho [Ashley91].

En Europa, los primeros trabajos sobre CBR los publican, a finales de los 80, Derek Sleeman, de la Universidad de Aberdeen, y Mike Keane, del Trinity College de Dublín. A principios de los 90 surge un grupo en la Universidad de Kaiserslautern, encabezado por Michael Richter y Klaus Althoff, que investigan en la aplicación del CBR a tareas de diagnóstico complejas. La Unión Europea financió en la década de los 90 dos proyectos, INRECA I y II, dedicados a investigar en el desarrollo de herramientas y metodologías para sistemas CBR, alrededor de los cuales surgieron y se consolidaron dos compañías: AcknoSoft en Francia y tecInno en Alemania.

4. Fundamentos del CBR

El CBR está inspirado, en gran medida, en el papel que juega el recuerdo en el razonamiento humano. Así pues, el CBR es un mecanismo de razonamiento que se basa en recordar situaciones o experiencias (casos) similares acontecidas en el pasado y almacenadas en una base de experiencias, y adaptar la lección extraída de ellas a la situación actual [Aamodt&Plaza94] [Althoff *et al.* 95] [Kolodner93] [Leake96] [Watson97]. Los dos pilares del enfoque del CBR son dos suposiciones fundamentales acerca de la naturaleza del mundo [Kolodner96]:

- Su regularidad. De situaciones similares se extraen conclusiones y se aprenden lecciones similares. En consecuencia, las conclusiones y lecciones que acompañan a experiencias previas pueden ser la base de las que correspondan a una situación nueva.
- La recurrencia de las experiencias. Es altamente probable que las situaciones futuras sean variantes de las actuales.

Admitiendo la suposición básica de que "problemas parecidos tienen soluciones parecidas" y que "los nuevos problemas son similares a problemas previamente resueltos", la resolución de problemas basada en casos saca partido de las relaciones entre dos tipos de similitud. Estos tipos de similitud se aplican a dos espacios diferentes, el espacio de la descripción

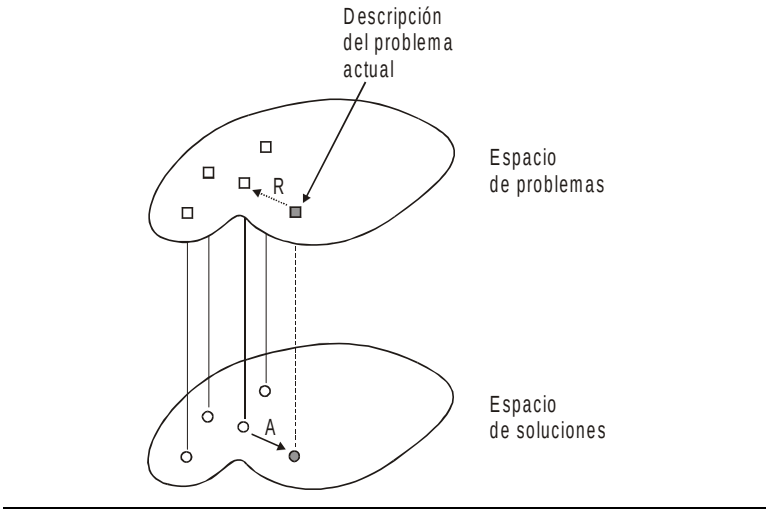


Figura 2-1. *Suposición básica en un sistema CBR.*

de los problemas y el espacio de las soluciones a los problemas. La Figura 2-1 muestra el papel que juegan dichas similitudes en el proceso de generación de soluciones. La suposición anterior depende directamente de las características que se utilicen para describir estos problemas en el espacio de descripción de los problemas. El sistema de CBR identifica un problema previo cuya descripción sea la más similar a la del nuevo en función de las características representadas. En concreto, de aquellas características más relevantes que serán identificadas como *índices* para guiar el proceso de recuperación. La solución del problema considerado más similar se utiliza como punto de partida para generar la solución al nuevo problema, solución que no deberá ser muy diferente a la inicial[1].

Fijémonos en que, en todo momento, hemos hablado de utilizar experiencias *similares*. La característica común a todos los sistemas de CBR es el hecho de llevar a cabo una localización aproximada de experiencias previas y una selección de la mejor de ellas en base a la similitud con la nueva situación. En consecuencia, la información con la que debe contar un sistema de CBR no es únicamente la almacenada en los casos. Se necesita también, al menos, conocimiento relativo al cálculo de la similitud. Y esto es así independientemente de la aplicación concreta del sistema CBR.

El siguiente apartado describe el ciclo de tareas que se deben resolver cuando se aplica CBR. Antes de describir cada una de estas tareas en los Apartados 4.3 al 4.6, el Apartado 4.2 analiza distintos tipos de conocimiento que podemos encontrar en los sistemas CBR.

4.1 El ciclo CBR

La resolución de problemas en CBR se realiza de la siguiente forma: ante la descripción de un nuevo problema –*consulta*– se recupera un caso previo parecido al problema actual. En base a las diferencias entre la descripción del caso pasado y la del problema actual, se adapta la solución del caso resuelto para obtener la solución del caso actual. A continuación se *revisa* la solución y se *aprende* el nuevo caso junto con la solución revisada. Son cuatro los procesos básicos en los que se apoya este modelo que se muestra en la Figura 2-2.

[1] Asumiendo una correspondencia uno-a-uno entre la descripción de los problemas y sus soluciones

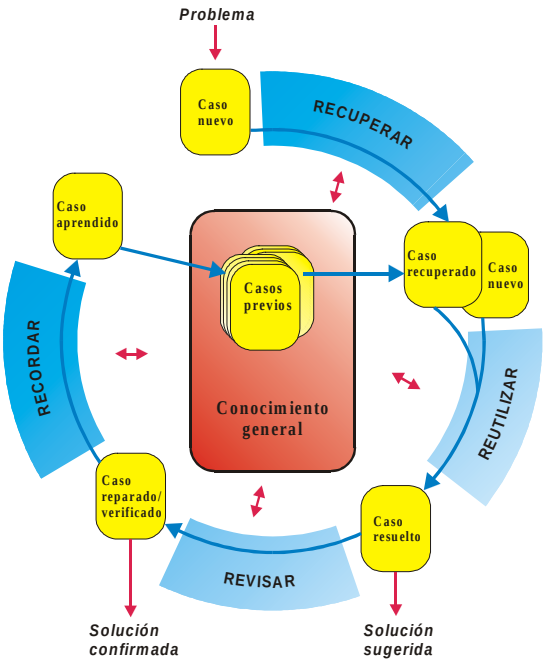


Figura 2-2. *El ciclo CBR [Aamodt&Plaza94].*

La recuperación de los casos similares. La construcción de formalismos de representación de los casos que faciliten el cálculo de la similitud junto con la eficiencia de la recuperación –ordenados por similitud– es una de las áreas de investigación más activas en CBR y en la que se han obtenido más resultados.

La *reutilización* o *adaptación* se hace necesaria cuando la solución recuperada no es directamente aplicable al problema en curso. La adaptación puede ir desde un simple ajuste de parámetros mediante la aplicación de ciertas fórmulas, hasta el uso de modelos complejos de comportamiento.

La fase de *revisión* se encarga de validar, y reparar en su caso, la solución propuesta. La fase de validación suele realizarse de manera externa al sistema o utilizando modelos más completos que aquellos que el sistema utiliza para la adaptación.

El *recuerdo* de nuevos casos es una parte importante de un sistema CBR donde, como ya se ha mencionado, el "aprendizaje continuado" es una de las ventajas fundamentales. A medida que aumenta el número de casos se plantean cuestiones de eficiencia –los procesos son más lentos cuando hay más casos– y es necesario ser más crítico a la hora de decidir qué casos se incluyen en el sistema. En la literatura reciente se describen técnicas de mantenimiento de bases de casos que eliminan los casos poco utilizados, o de identificar familias de casos relacionados para así mantener sólo los casos que aporten información al sistema.

La forma en que se representan los casos, cómo se determina la similitud, cómo se realiza la adaptación y cómo se decide cuántos casos hay que almacenar para conseguir una cobertura suficiente del dominio de aplicación, son cuestiones que están íntimamente relacionadas entre sí. Por ejemplo, mientras más capacidades de adaptación tenga el sistema, menos casos hay que almacenar. Además, si hay más conocimiento de adaptación entonces es necesario menos conocimiento de similitud.

Las aplicaciones agrupadas bajo el paradigma CBR se clasifican en dos grandes grupos [Kolodner93]: clasificación e interpretación de situaciones; y resolución de problemas. En la interpretación el aspecto fundamental es decidir si una nueva situación puede o no ser tratada como otras situaciones previas, en base a las similitudes y diferencias entre ellas. En la resolución de problemas el objetivo es la construcción de una solución para un caso adaptando las soluciones de casos previos. En el ámbito de los sistemas CBR de clasificación e interpretación destacan las aplicaciones de servicios de atención al cliente, de diagnóstico y resolución de fallos, de predicción y valoración y de comercio electrónico. Por lo que se refiere a los sistemas CBR de resolución de problemas, sus aplicaciones se restringen, casi exclusivamente, al ámbito académico en dos áreas clásicas de IA: planificación y diseño.

Aunque esta distinción no siempre está clara en la práctica, ya que algunos sistemas pueden clasificarse en ambos grupos, la presencia y complejidad de la fase de adaptación marca en gran medida la distinción entre los tipos de sistemas. En particular, la adaptación se obvia en muchos sistemas, o se deja en manos de los usuarios debido a que requiere modelos del dominio más elaborados y, en cierto modo, va en contra del espíritu del CBR que aboga por aliviar los procesos de adquisición de conocimiento. Es muy significativo que la mayoría de los sistemas comerciales obvien o traten de forma bastante trivial el problema de la adaptación. De esto se deduce que, aunque el ciclo de la Figura 2-2 es un marco general donde cabe cualquier sistema CBR, no todos incluyen las cuatro fases del ciclo.

Antes de describir con más detalle cada una de estas fases, el siguiente apartado resume los distintos tipos de conocimiento que se suelen incluir en los sistemas CBR.

4.2 Conocimiento en un sistema CBR

El núcleo de conocimiento de un sistema CBR está formado por la base de casos. Sin embargo, en la Figura 2-2 vemos que el conocimiento almacenado en los casos no es el único con el que cuenta un sistema CBR, ya que los casos aparecen embebidos en algún tipo de conocimiento *general*. Este conocimiento se entiende como cualquier información del sistema que no se encuentra recogida en forma de casos y que interviene en los distintos procesos del ciclo CBR, pudiendo variar su importancia en distintas aproximaciones al CBR.

Durante varios años, la comunidad de CBR ha desarrollado diversos métodos para resolver los procesos del ciclo CBR en distintos dominios, para distintos tipos de aplicaciones, y haciendo uso de distintos tipos de conocimiento. El conocimiento involucrado en los procesos del ciclo CBR de la Figura 2-2 ha sido clasificado en términos de los siguientes contenedores (*knowledge containers* [Richer95]):

- El conocimiento ontológico o vocabulario del dominio, que incluye la terminología utilizada en los casos, y los atributos y los valores permitidos para ellos.
- Los casos, que son el corazón del CBR, descritos con esa terminología.
- Las medidas de similitud y otros criterios de búsqueda. Este conocimiento puede estar representado de forma explícita –en forma de jerarquías conceptuales que expliciten la relación entre los elementos del dominio, o de parámetros para pesar los atributos– puede estar oculto dentro del algoritmo de recuperación.
- El conocimiento de adaptación o de transformación de la solución. Algunos sistemas usan conocimiento para verificar la utilidad de un caso elegido y adaptado, por ejemplo aplicando los pasos de la solución del caso elegido al problema consulta.

Adicionalmente, se han propuesto extensiones. Por ejemplo, en [Paterson *et al.* 01] se propone añadir un contenedor adicional relativo al conocimiento de mantenimiento.

Todos los sistemas CBR comparten la representación explícita de los casos. Sin embargo, el resto del conocimiento puede formar parte, de forma implícita, de los algoritmos que resuelven los procesos CBR o estar representado de forma explícita como un depósito de conocimiento general sobre el dominio que es utilizado por los algoritmos. Realmente este conocimiento general del dominio que da soporte a los procesos CBR puede ser de muy distinta naturaleza, variar en el grado de completitud y representarse de diversas formas en las que no profundizaremos: modelos causales, modelos de comportamiento, heurísticas, restricciones, modelos terminológicos y jerarquías conceptuales [Aamodt00].

Además del razonamiento con casos –aprovechando cierto conocimiento adicional- existen algunos sistemas mixtos que integran distintos tipos de razonamiento. Por ejemplo, uno de los primeros fue el sistema CASEY [Koton89] que integra razonamiento causal basado en un modelo causa-efecto para diagnosticar enfermedades del corazón. Como entrada toma los síntomas del paciente y produce una red causal de posibles estados internos que podrían haber conducido a esos síntomas. Cuando aparece un caso nuevo CASEY intenta encontrar casos de pacientes con síntomas parecidos. Si los encuentra, CASEY adapta el diagnóstico recuperado en función de las diferencias entre los síntomas del caso nuevo y el viejo. Si el CBR falla y no es capaz de proporcionar un diagnóstico, CASEY ejecuta el método basado en modelo para resolver el problema y almacena la solución como un nuevo caso para uso futuro. También en el dominio médico, el sistema BOLERO [Lopez&Plaza93] integra razonamiento basado en reglas y basado en casos. BOLERO construye un plan de diagnóstico a partir de la información conocida sobre un paciente. El componente basado en reglas tiene conocimientos sobre enfermedades específicas, mientras que el componente CBR tiene conocimiento sobre el cuidado de los pacientes. Otro ejemplo es el sistema CREEK [Aamodt91] [Aamodt93] que integra resolución de problemas con aprendizaje máquina. CREEK integra distintos tipos de conocimiento y de razonamiento, incluyendo una biblioteca de casos, conexiones entre casos y diagnósticos, reglas heurísticas y modelos "profundos" en una estructura de conocimiento unificada. Las reglas o los modelos "profundos" pueden usarse para resolver problemas de manera autónoma cuando el razonamiento CBR falla.

Realmente, aunque el término KI-CBR en el hemos enmarcado nuestro trabajo podría dar cabida a las aproximaciones anteriores, basadas en la representación explícita de conocimiento complementario al de los casos, no lo utilizamos en este sentido. En nuestro trabajo utilizaremos el término KI-CBR para denominar a la línea de trabajo que estudia la representación explícita de conocimiento taxonómico –principalmente terminológico– del dominio y su impacto en los procesos CBR, particularmente en aspectos relacionados con la valoración de la similitud entre elementos del dominio -tanto para recuperar casos similares como para adaptar casos por sustitución de ciertos elementos por otros parecidos- y en la organización de los casos en torno a este tipo de conocimiento.

Los siguientes apartados describen las distintas componentes de conocimiento que pueden intervenir en un sistema CBR, comenzando naturalmente por los casos y agrupando el resto en los epígrafes de conocimiento terminológico del dominio, conocimiento de similitud y organización de la base de casos.

4.2.1 Los casos

El principal activo de un sistema CBR es la base de casos. Pero, ¿qué es un caso? Según Janet Kolodner y David Leake [Kolodner&Leake96]:

> "*Un caso es un fragmento contextualizado de conocimiento que representa una experiencia que enseña una lección importante para conseguir los objetivos del razonador*"

En esta breve definición aparecen varias ideas importantes. Un caso contiene información útil en un contexto concreto; el problema es identificar los atributos que caracterizan al contexto y detectar cuándo dos contextos son similares. Un caso es una experiencia que enseña algo, ya que puede haber experiencias que no aporten nueva información al sistema, lo cual plantea el problema de identificar cuándo dos casos superficialmente distintos, contienen información redundante entre sí. Y, por último, lo que el caso enseña es relativo a unos determinados objetivos, y, por lo tanto, un caso puede ser útil o inútil dependiendo del objetivo, o resultar útil para distintos fines.

En su formulación más general un caso se compone de:

- La descripción de un problema, ya sea la situación a interpretar, el problema de planificación a resolver o el artefacto a diseñar. Es la parte del caso que codifica el estado del mundo en el que comienza el razonamiento. Puede estar constituido por los objetivos, las restricciones impuestas sobre los objetivos y cualquier otro tipo de característica que limite el contexto al que es aplicable la solución del caso.
- La descripción de una solución asociada al caso representa el conocimiento necesario para alcanzar los objetivos que aparecen en la descripción del caso, teniendo en cuenta las restricciones especificadas y el resto de características contextuales, si las hay. Las soluciones pueden adoptar formas muy diversas, desde el artefacto a diseñar, si se trata de un problema de diseño, hasta un plan de actuación, si se trata de un problema de planificación. Un enfoque alternativo consiste en representar no objetos o acciones concretos (soluciones reales) sino trazas que indican cómo se ha llegado a las soluciones. En ocasiones puede resultar interesante representar los dos tipos de solución.
- El resultado de aplicar la solución. Teniendo en cuenta que una determinada solución puede fallar, interesa guardar no sólo las soluciones que funcionan sino también aquellas otras que fallaron, ya que ambas contienen información útil que permitirá repetir las soluciones exitosas, y evitar la repetición de las fallidas. El resultado del caso especifica las restricciones producidas como consecuencia de la adopción de la solución propuesta para el problema especificado.

En algunas propuestas se incluye una componente adicional: la justificación de un caso. La justificación está relacionada con las soluciones en forma de traza y aclara o explica por qué se han tomado las decisiones que se han tomado. El objetivo de las justificaciones es doble. Por una parte, ayudan durante la evaluación de la solución. Si algunas partes de una solución están justificadas, una forma de evaluar si la solución inicial es válida en la nueva situación es examinar si las propiedades de las partes justificadas de la solución inicial también están presentes en el nuevo caso –pudiéndose así aplicar la justificación a las partes correspondientes en la nueva solución. Por otra parte, las justificaciones guían en la selección de los índices de la base de casos. Las justificaciones indican qué aspectos del mundo del que se han extraído los casos son relevantes para las decisiones tomadas.

Dentro de esta estructura general, podemos encontrar sistemas donde los casos sólo contienen problema y solución, o incluso aquellos otros donde no se hace una distinción explícita entre las partes de un caso, sino que cada caso está descrito por un conjunto de atributos y una consulta sólo contiene una parte de ellos, de forma que el problema son los atributos incluidos en la consulta y la solución los que no están. Este es el esquema típico por ejemplo en las aplicaciones de diseño.

Las formas de representar la solución, la justificación y el resultado varían según el dominio de aplicación del sistema de CBR y la aproximación utilizada. En [Althoff *et al.* 95] distinguen dos grandes grupos de representaciones para los casos:

- Representaciones planas. En este tipo de representación se define una serie de atributos, cada uno con un conjunto de posibles valores de tipos simples –cadenas de caracteres, números, símbolos. En general no se define relación alguna entre los atributos ni entre sus valores, esto es, si utilizamos *n* atributos en la representación, un caso vendrá descrito por la *n*-tupla de valores de los atributos del caso en cuestión.

- Representaciones estructuradas. También aquí se utilizan listas de atributos y valores asociados, pero a diferencia de las representaciones planas, los valores pueden ser objetos que a su vez tienen atributos. De esta forma se consiguen descripciones más expresivas en las que se definen relaciones entre los atributos y/o entre los valores. Para implementar este tipo de descripciones se suele utilizar cálculo de predicados, redes semánticas, lenguajes basados en marcos, lenguajes orientados a objetos y lógicas descriptivas (DLs), entre otros.

  Utilizando terminología basada en objetos, en [Bergmann&Stahl98] se presenta un esquema general para la representación estructurada de casos, que es aplicable (salvo terminología y variaciones específicas) a otras implementaciones. Según el esquema general, los casos se representan como objetos que son instancias de alguna clase que determina su estructura, es decir, sus atributos y los tipos del valor de relleno de estos atributos. Además, típicamente las clases a las que pertenecen los objetos se organizan en una jerarquía de clases con herencia. Se distingue entre dos tipos de atributos. Los atributos *simples* son aquellos cuyos rellenos son valores de alguno de los tipos predefinidos, por ejemplo números o símbolos. Los atributos *relacionales* son aquellos cuyos rellenos son objetos de otras clases de la jerarquía, y son los que permiten representar estructuras complejas para los casos. Además, los casos pueden tener estructuras distintas ya que si el relleno de un atributo relacional debe ser un objeto de una cierta clase, será válido cualquier relleno que sea instancia de cualquiera de sus subclases que pueden introducir estructuras distintas.

Además de los dos tipos anteriores existen otros, por ejemplo, la aproximación textual que utiliza el lenguaje natural como mecanismo de representación de casos, variando en la utilización de distintas estructuras de texto, o aproximaciones multimedia que incluyen la representación de gráficos, vídeos y/ sonidos. Por supuesto el tipo y la estructura de representación de casos influyen en los procesos encargados de manejar dichos casos. Por ejemplo, la valoración de la similitud es muy sencilla cuando manejamos representaciones planas, se convierte en un proceso de complejidad creciente con la estructura de los casos [Bunke&Messmer93] [Matuscheck&Jantke97] [Bridge98], y requiere técnicas de procesamiento del lenguaje natural cuando se usan representaciones textuales [Brüninghaus&Ashley01].

El uso de representaciones estructuradas presenta varias ventajas. Por un lado la expresividad, ya que ofrece una forma natural de describir objetos compuestos cuya representación mediante pares atributo-valor causará algunos problemas típicos, por ejemplo qué hacer con los atributos que son irrelevantes para un determinado caso o los valores no aplicables, que provocarán otros problemas en la valoración de la similitud entre casos. Además, la representación estructurada de casos ofrece la posibilidad de considerar las subpartes de los casos como casos en sí mismos que pueden ser guardados, recuperados y utilizados para resolver los (sub)problemas de los nuevos casos; además un caso puede resolverse utilizado subpartes de distintos casos recuperados.

La tendencia actual es hacia sistemas CBR asociados con problemas que requieren casos con mayor nivel de contenido y con estructuras más complejas [Aha00]. Sin embargo, si las características del sistema permiten asumir que los casos se describan como conjunciones de pares atributo-valor supone un gran número de simplificaciones en cuanto a representación y recuperación.

Como ya hemos dicho, además de los casos generalmente un sistema CBR incluye otros tipos de conocimiento general. Aunque el origen, tipo y uso de este conocimiento varía considerablemente entre distintos sistemas y tipos de CBR, existe acuerdo en que como mínimo se debe incluir conocimiento de similitud que permita, ante un nuevo problema, seleccionar el caso, o casos, más similares según ciertos criterios que permiten su reutilización en la resolución del nuevo problema. Aunque este conocimiento de similitud puede aparecer de forma implícita en los algoritmos (ver Apartado 4.2.3), nosotros estamos interesados en una línea de investigación en sistemas KI-CBR que propone procesos de razonamiento que hacen uso de un modelo de conocimiento general terminológico sobre el dominio como se describe brevemente en el apartado siguiente y más en profundidad en los Capítulos 4, 5 y 6.

4.2.2 Conocimiento terminológico del dominio

Como hemos visto, existen distintos tipos de conocimiento que los sistemas CBR pueden utilizar como conocimiento de soporte para llevar a cabo sus procesos de razonamiento. En este epígrafe, y en esta tesis, estamos interesados en el conocimiento terminológico en forma de taxonomías, principalmente jerarquías conceptuales que representan el vocabulario del dominio, y el impacto de este tipo de conocimiento en los procesos CBR. Lógicamente, para que esta aproximación tenga sentido, este conocimiento terminológico representa el vocabulario utilizado en los casos, por ejemplo, qué atributos pueden aparecer y cuáles son los valores permitidos para ellos.

Como primer ejemplo de este tipo de conocimiento, citamos la representación que hace el sistema CREEK del conocimiento del dominio en una única red semántica de conceptos y relaciones representados mediante marcos organizados en una estructura taxonómica. El sistema utiliza este conocimiento del dominio como modelo de soporte para los procesos CBR y para generar explicaciones de los resultados que genera el sistema. Relacionada con la red semántica de conceptos y relaciones de CREEK, el sistema CRASH [Brown93] hace una reformulación de los procesos del CBR de valoración de la similitud y la adaptación como procesos indirectos de búsqueda de información en una estructura con una representación explícita del "conocimiento del mundo".

Existen muchas otras aproximaciones como [Plaza95] [Napoli *et al.* 96] [Bergmann *et al.* 96] [Kamp96] [González-Calero97] [Salotti&Ventos98] [Bergmann&Stahl98] [Gómez-Albarrán00] consideran que los casos se organizan en una estructura conceptual de clasificación donde los conceptos (clases) más específicos están clasificados bajo los más generales, y los casos que se recuperan son las instancias de los conceptos de la jerarquía. Esta estructura es adecuada para varios formalismos de representación de conocimiento, como los sistemas basados en objetos, los sistemas de marcos o las lógicas descriptivas.

De lo anterior se extrae una idea importante a tener en cuenta cuando este tipo de conocimiento taxonómico está presente: ¿La taxonomía de conocimiento terminológico del dominio es adecuada como estructura para organizar la base de casos?

En general, la terminología representa de conocimiento del dominio que esta disponible a priori y que incluye el vocabulario con el que se definen los casos organizado de forma taxonómica. En este sentido, la taxonomía de términos del dominio organiza, de una forma indi-

recta, a los casos que se describen utilizando dichos términos e incluye conocimiento de similitud sobre ellos como veremos en el Apartado 4.2.3. Es decir, las estructuras de conocimiento terminológico se pueden aprovechar como estructuras en la que los casos quedan inmersos de forma natural al utilizar para su descripción vocabulario de dicha terminología. En este caso la jerarquía conceptual también incluye conocimiento de similitud pero no es una estructura de organización optimizada para ninguna medida concreta.

Sin embargo, la taxonomía con conocimiento terminológico puede no ser utilizada como estructura de organización sino con otros objetivos. Si esta es la situación, la estructura en la que se clasifican los casos se computa a posteriori, como conocimiento de soporte para el algoritmo de recuperación, y estará integrada con el conocimiento terminológico del dominio únicamente por el uso de un vocabulario común. Estas estructuras se computan para explicitar una medida de similitud y se añaden únicamente como una forma de mejorar la eficiencia del proceso de recuperación frente a la recuperación por cómputo de similitud lineal sobre la base de casos [Porter89]. Este tipo de estructuras de organización se describen en el Apartado 4.2.4.

Aunque el conocimiento terminológico del dominio se ha utilizado principalmente como conocimiento de organización de casos y valoración de la similitud, también se han utilizado como soporte de otros procesos CBR. Por ejemplo, igual que el sistema CREEK, otros trabajos [Kass *et al.* 86] [Leake92] utilizan este tipo de conocimiento para la generación de explicaciones y la justificación del razonamiento. Respecto a la adaptación de casos es sin duda el proceso de mayor complejidad y es, por tanto, el que requiere más conocimiento adicional. Aunque existen varias aproximaciones, como los sistemas combinados de reglas y casos, en los que la adaptación utiliza modelos complejos de comportamiento, estamos más interesados en otra línea de trabajo en la que se aprovecha la estructura terminológica de conocimiento del dominio como conocimiento de adaptación, sobre el que se pueden llevar a cabo procesos de reinstanciación [Hammond89] o búsquedas [Kass90]. Estas aproximaciones se describen en el Apartado 4.4.2.

4.2.3 Conocimiento de similitud

La recuperación del caso más adecuado para una cierta situación requiere ser capaz de llevar a cabo un proceso de *encaje parcial* entre los casos. En general este proceso no es perfecto porque por un lado las características que describen los casos pueden no ser exactamente las mismas, y por otra parte, los valores de las características en general tampoco coinciden exactamente, aunque pueden ser más o menos parecidos. Por esta razón la aproximación normal es definir algún modo de medir la similitud entre los casos.

La similitud debe modelar la utilidad de un caso previo para resolver el problema actual. Sin embargo, este planteamiento es en principio paradójico: dos casos se deben considerar similares si la solución de uno permite obtener fácilmente la solución del otro; sin embargo esto sólo se puede saber *después* de que se haya intentado reutilizar la solución, y para ello es necesario haber elegido el caso más similar. Esta paradoja se resuelve mediante la suposición básica en los sistemas CBR de que si dos problemas tienen descripciones parecidas, entonces sus soluciones también lo serán (Figura 2-1).

Dependiendo del tipo de información que se represente, existen distintas técnicas estándar para la definición de medidas de similitud. Considerando que un caso viene descrito por

	Univaluado	Multivaluado	
Simbólico	$\begin{cases} 0, & \text{si } a \neq b \\ 1, & \text{si } a = b \end{cases}$	$\frac{card(a \cap b)}{card(a \cup b)}$ $\frac{card(a \cap b)}{\min(card(a), card(b))}$	$\frac{card(a \cap b)}{card(O)}$ $\frac{card(a \cap b)}{\max(card(a), card(b))}$
Numérico	$1 - \frac{\lvert a - b \rvert}{long(I)}$	$\frac{long(a \cap b)}{long(a \cup b)}$ $\frac{long(a \cap b)}{\max(long(a), long(b))}$	$1 - \frac{\lvert a_c - b_c \rvert}{long(O)}$ $\frac{long(a \cap b)}{\min(long(a), long(b))}$

siendo O el conjunto de valores posibles, long(I) la longitud del intervalo I, y ac el punto central del intervalo a.

Figura 2-3. *Funciones de similitud local sim(a, b).* [Althoff *et al.* 95]

un conjunto de atributos[2], la aproximación más básica para el cálculo de la similitud consiste en contabilizar los valores iguales en los atributos comunes de los casos a comparar:

```
leer(C1, C2);
todosAtributos := atributos(C1) ∪ atributos(C2);
tamaño := card(todosAtributos);
valoresIguales := { a | C1.a = C2.a };
iguales := card(valoresIguales);
devolver(iguales/tamaño);
```

Sin embargo, las cosas no son tan simples. En primer lugar, se suele distinguir entre funciones de *similitud local*, que calculan la similitud entre los valores del mismo atributo en dos casos distintos, y funciones de *similitud global*, que combinan los resultados de aplicar ciertas funciones de similitud local a todos los atributos de los casos que estamos comparando. Además, este esquema sólo es adecuado para representaciones planas. El uso de representaciones estructuradas añade complejidad adicional al cómputo de la similitud.

En cuanto a la definición de medidas de similitud local, hay que considerar qué tipo de atributos estamos comparando: números, símbolos o estructuras complejas; y si cada atributo puede tomar un valor –univaluada– o varios simultáneamente –multivaluada–. Para atributos simbólicos y numéricos existen una serie de funciones estándar, algunas de las cuales se muestran en la Figura 2-3. Otra estrategia habitual para definir medidas de similitud entre atributos simbólicos consiste en construir una *tabla de similitudes* donde se recoja explícitamente la similitud entre cada par de valores. A los atributos simbólicos ordenados se les pueden aplicar las mismas funciones que a los atributos numéricos, convirtiéndolos en números mediante el ordinal. En representaciones más complejas que implican algún tipo de jerarquía de valores se pueden utilizar otras funciones más sofisticadas que tengan en cuenta dicha organización.

Respecto a las funciones de similitud global la forma general es:

[2] No consideraremos otros tipos de representaciones de casos, por ejemplo, las textuales.

$\frac{1}{p}\sum_{i=1}^{p} sim_i(a_i, b_i)$	bloques	$\frac{1}{p}\left[\sum_{i=1}^{p}\left[sim_i(a_i, b_i)\right]^r\right]^{1/r}$	minkowski
$\sum_{i=1}^{p} w_i \cdot sim_i(a_i, b_i)$	bloques ponderada	$\left[\sum_{i=1}^{p} w_i \cdot \left[sim_i(a_i, b_i)\right]^r\right]^{1/r}$	minkowski ponderada
$\frac{1}{p}\left[\sum_{i=1}^{p}\left[sim_i(a_i, b_i)\right]^2\right]^{1/2}$	euclídea	$\max_{i=1}^{p}\left[w_i \cdot sim_i(a_i, b_i)\right]$	máximo
$\frac{\sum_{i=1}^{P} sim_i(a_i, b_i)}{P}$	Media Aritmética	$\min_{i=1}^{p}\left[w_i \cdot sim_i(a_i, b_i)\right]$	mínimo

$$donde \quad \sum_{i=1}^{p} w_i = 1$$

Figura 2-4. *Funciones de similitud global SIM(A, B).* [Althoff *et al.* 95]

$$SIM(A,B) = F(sim_1(a_1,b_1), sim_2(a_2,b_2), \ldots, sim_p(a_p,b_p))$$

donde A y B son dos casos, $F : [0,1]^p \rightarrow [0,1]$, y *sim*i es la función de similitud local que se aplica para comparar los valores del atributo i.

En la Figura 2-4 se muestran algunas funciones estándar para el cálculo de la similitud global. Es de resaltar que todas ellas son, en esencia, funciones geométricas ya que computan distancias en el espacio métrico definido por las funciones de similitud local.

Según [Althoff *et al.* 95] las propiedades deseables de una medida de similitud son la *reflexividad*, todo caso es similar a sí mismo, y la *simetría*, el caso A es igual de similar a B, que B de A. La simetría se cumplirá siempre que la similitud se defina como la inversa de la distancia. En algunos trabajos, como el de [Bergmann&Stahl98], se proponen refinamientos que señalan el interés de utilizar funciones de similitud no simétricas, dependiendo de si la descripción es de un caso o de una consulta. En general, una medida de similitud no cumplirá la propiedad transitiva, ya que un caso A puede ser similar a otro B (por unas ciertas características) y B ser similar a un tercer caso C (por otras características distintas) de forma que A no sea similar a C.

4.2.3.1 Medidas de similitud para comparar objetos estructurados

El cálculo de la similitud depende, lógicamente, del tipo de representación utilizado para los casos. Con el uso de representaciones estructuradas surge la necesidad de un cambio en las medidas de similitud utilizadas con representaciones de casos simples que supone un aumento del coste del cómputo de la similitud.

Se pueden clasificar las medidas de similitud entre casos estructurados según el tipo del valor generado como resultado. Se puede computar un valor simple, por ejemplo, booleano o numérico, o utilizar una medida de *similitud estructural* que genera una estructura compleja

que represente de forma explícita cuál es la similitud [Plaza95] [Matuschek&Jantke97] [Bridge98] [Salotti&Ventos98].

Medidas que generan un valor simple

En el contexto de uso de un esquema estructurado de representación, una medida de similitud adecuada debe permitir comparar dos casos con distinta estructura, por ejemplo, dos objetos que pertenecen a clases distintas.

En [Osborne&Bridge97] se propone una simplificación para computar la similitud entre dos objetos estructurados que se basa en definir funciones de proyección que recorren el grafo de representación del objeto caso hasta las hojas para obtener un valor simple y poder utilizar las funciones de similitud clásicas para los esquemas de representación con vectores atributo-valor que hemos descrito en el apartado anterior. Sin embargo, esta simplificación no tiene en cuenta todo el conocimiento en forma de objetos intermedios que intervienen en la representación del caso. En general, para computar la similitud entre dos objetos estructurados (o1 y o2), uno representando un caso o una parte suya, y el otro representando una consulta o una parte suya, se usa una aproximación clásica recursiva. Para cada atributo simple común a ambos objetos, se utiliza una medida de similitud *local* que calcula la similitud entre los dos valores del atributo. Para cada atributo relacional se usa una medida de similitud *global* que compara de forma recursiva los dos subobjetos. Después todos los valores de similitud se combinan -por ejemplo usando una suma ponderada- para obtener el valor de similitud final [Bergmann&Stahl98]. Es decir,

$$SIM(o1, o2) = \begin{cases} f\,(SIM(o1.A1, o2.A1), .., Sim(o1.An, o2.An)) & \text{si o1 y o2 son objetos estructurados} \\ simLocal(o1, o2) & \text{si o1 y o2 son valores} \end{cases}$$

Donde oi.Ai representa el relleno del objeto oi en el atributo Ai y donde las similitudes se computan sobre los atributos Ai que son comunes a o1 y a o2. Si Ai es un atributo simple la forma de computar la similitud local depende del tipo de los rellenos del atributo Ai. Si Ai es un atributo relacional el cómputo de similitud entre los rellenos genera una llamada recursiva que computa la similitud entre los dos objetos.

Esta aproximación clásica tiene un inconveniente cuando existe un modelo terminológico general sobre el dominio, ya que en la medida de similitud sólo contribuye la información sobre los atributos y valores de los objetos y no tiene para nada en cuenta las clases a las que pertenecen los objetos. Es decir, deberíamos tener en cuenta cómo la jerarquía de clases del modelo del dominio *influye en la similitud*, ya que intuitivamente los objetos cercanos en la jerarquía son más similares que los lejanos. Por esto, la medida de similitud cuando se dispone de un modelo de conocimiento del dominio en forma de taxonomía de clases, debería combinar la similitud debida a la cercanía de las clases de los objetos comparados con la similitud debida a los valores de sus atributos [Bergmann&Stahl98]. Además, hay que considerar que estas dos componentes no son del todo independientes, porque los valores de los atributos determinan la posición del individuo en la taxonomía.

En muchos sistemas CBR se restringe el cómputo de la similitud a la comparación de objetos con la misma estructura y que pertenecen a las mismas clases. Esta aproximación no aprovecha la flexibilidad que proporciona la representación orientada a objetos y no es válida en situaciones que permitan herencia múltiple, donde aunque dos objetos pertenezcan a la misma clase no se puede garantizar que tengan la misma estructura. Por tanto, es importante considerar cómo determinar la similitud entre dos objetos que pertenecen a clases distintas. En el trabajo de [Bergmann&Stahl98] se propone una solución a este problema en la que, para computar la similitud entre dos objetos, se combinan dos factores: la similitud dentro de

la clase (*intra class similarity*) y la similitud entre clases (*inter class similarity*). Los valores obtenidos en cada una se multiplican para computar el valor de similitud final.

El primer factor (*intra class*) se basa en la similitud entre los valores de los atributos que son comunes a los dos objetos comparados. Se corresponde con el esquema clásico anterior. Es similitud "dentro" de una clase porque la clase más específica que es común a los dos objetos, es decir, a la que ambos pertenecen, determina una parte de la estructura que es común entre los dos objetos y por tanto el conjunto de atributos que son comunes a ambos

El segundo factor (*inter class*) representa la similitud de dos objetos independientemente de los valores de sus atributos. Este factor sólo dependerá de la posición en la jerarquía que ocupen las clases a las que pertenezcan los objetos que estamos comparando. La jerarquía de clases codifica cierto conocimiento sobre la similitud de los objetos que contiene de forma que cuanto más profundo bajemos en la jerarquía más detalles e información damos, es decir, cuanto más profunda esté la clase más específica común a ambos objetos, más similares son los objetos. Podemos observar que esta componente nos ofrece una medida de similitud *cualitativa* y *no cuantitativa*. Por esto, para obtener un valor numérico de similitud entre clases, añade conocimiento adicional a la jerarquía. En [Bergmann&Stahl98] se propone anotar los nodos con un valor de similitud que es creciente con su profundidad en la jerarquía. Una posible ventaja que justifica esta aproximación es que estos números están precomputados para todos los conceptos/clases de la jerarquía y por tanto es eficiente ya que no hay que computar valores de similitud durante la interacción con el usuario. La desventaja proviene del coste de anotar los nodos y de la falta de flexibilidad del uso de números prefijados en fase de diseño.

Además de la aproximación anterior, consistente en anotar los valores de similitud en los nodos de la taxonomía, existen otras aproximaciones que se han considerado en la literatura. Aquellas, como [Plaza95] [Salotti&Ventos98] que consideran que la clase o concepto común más específico al que pertenecen los dos objetos, o LCS (*least common subsummer*) en terminología de DLs [Cohen *et al.* 92], puede ser directamente un término de similitud que se puede mostrar al usuario y además de determinar "cuánto" de similares son los dos objetos podremos decir en "qué" cosas son similares. Estas aproximaciones se detallan en el apartado siguiente.

Otros trabajos proponen alternativas relacionadas con la propuesta de Bergmann en el sentido de que transforman la medida cualitativa en un valor numérico que interviene en la similitud final. Por ejemplo, en [González *et al.* 99b] se obtiene el número computando el coseno de los vectores definidos con los conceptos de los que cada individuo es instancia. LAUD [Armengol&Plaza01] es una medida de distancia para estimar la similitud de casos relacionales representados como *términos de características* [Plaza95].

Medidas que generan un valor estructurado

Las medidas descritas en el apartado anterior computan un valor numérico que representa la similitud entre dos casos estructurados. En este apartado describimos otra línea de trabajo que propone utilizar medidas de similitud estructural que, en vez de un valor simple, computen una estructura compleja que represente de forma explícita cuál es la similitud entre los casos estructurados.

Por ejemplo, el trabajo presentado en [Bridge98] propone un marco para definir medidas de similitud que obtienen como resultado un retículo. Este marco es adecuado para cualquier esquema de representación estructurada sobre la que se pueda definir un orden parcial, y permite definir medidas de similitud que computen estructuras numéricas, es decir, un retícu-

lo sobre números con un orden, o declarativas, utilizando un retículo sobre los conjuntos de las propiedades que comparten, con el orden de inclusión entre conjuntos.

El trabajo de [Matuschek&Jantke97] propone el cómputo de expresiones de similitud para conjuntos de casos en contraste con la aproximación clásica que se refiere únicamente a pares de casos. Dado un operador de similitud σ habrá que definir un operador de similitud generalizado para conjuntos de casos que será independiente del orden de procesamiento de esos casos cuando σ tenga las propiedades de asociatividad y conmutatividad. Desde otra perspectiva propone sustituir los axiomas de asociatividad y conmutatividad por el de maximalidad para que los conceptos de similitud reflejen la subestructura común mayor respecto al orden usado. En su axiomatización concluyen que para que exista un operador de similitud estructural maximal es necesario y suficiente que el conjunto sobre el que se define el operador sea un semiretículo. Este resultado motiva la búsqueda de formalizaciones de los dominios donde el conjunto de todas las representaciones potenciales de casos es al menos un semiretículo con respecto a algún orden parcial elegido adecuadamente que refleje las relaciones estructurales.

Esta misma idea se utiliza en otros trabajos que utilizan como orden la relación de subsunción entre conceptos y proponen medir la similitud utilizando el LCS (*least common subsummer*) como estructura que representa la similitud entre cada par de casos. La aproximación original de E. Plaza [Plaza95] se basa en computar términos que representan la similitud entre casos mediante la *antiunificación* de los términos de características que se usan para representarlos. La antiunificación entre dos términos - que representan los casos- proporciona otro término cuya descripción representa la similitud entre dichos términos. La antiunificación de dos términos de características obtiene las propiedades comunes a ambos términos, es decir, la generalización más específica que contiene a ambos términos. La estrategia para la selección de los casos recuperados se basa en ordenar mediante clasificación los términos de similitud entre la consulta y cada caso de la biblioteca obtenidos por antiunificación.

El trabajo de [Plaza95], y como ampliación el de [Salotti&Ventos98], considera que la clase o concepto común más específico al que pertenecen los dos objetos, puede ser directamente un término de similitud que se puede mostrar al usuario y además de determinar "cuánto" de similares son los dos objetos podremos decir en qué cosas son similares.

La aproximación de [Plaza95] plantea una utilidad adicional de los términos de similitud. En concreto, utilizarlos para extraer de la base de casos conocimiento relativo a qué características son más importantes para comparar dos casos. Ya que se dispone de una definición simbólica de la similitud entre un problema actual y un precedente mediante un término de similitud, se puede valorar si el término de similitud incluye aspectos relevantes para la tarea que queremos resolver o a otros aspectos. Aunque existen varias alternativas para valorar la importancia de un término de similitud ellos proponen un criterio basado en el poder de discriminación del termino de similitud, que será evaluado mediante una función de entropía. Es decir para evaluar la importancia de la similitud entre una consulta y un caso recuperado valoran cuál es el poder de discriminación del término de similitud entre ellos, con respecto a todos los precedentes en la base de casos que resuelven la tarea actual.

Para computar la similitud entre la consulta Q y un caso C_i calculan el término de similitud entre ellos y lo clasifican en la jerarquía de subsunción, para obtener el conjunto D de todos los casos clasificados bajo ese término. Es decir, los casos que comparten las mismas características compartidas por Q y C_i. En el siguiente paso se comprueba si todos los casos

de D comparten una misma interpretación[3] o no, y evalúa el nivel de discriminación de la información compartida entre los casos. La forma de medir la importancia de un término de similitud se basa en medir la entropía del conjunto D respecto a los posibles valores de interpretación.

En [Salotti&Ventos98] se presenta una idea similar utilizando un sistema de lógica descriptiva como formalismo. Los casos almacenados se representan como *individuos* de la lógica descriptiva y están indexados por *conceptos* índices organizados automáticamente en una taxonomía según la relación de subsunción. Los índices de los casos se eligen con ayuda de un experto (indexación manual) para que sean relevantes con respecto a la tarea que va a llevar a cabo el sistema. La similitud entre dos casos (A y B) se caracteriza por un concepto computado (SIM(A,B)) que corresponde al LCS, o concepto más específico que los subsume a ambos. La diferencia entre dos casos se caracteriza por otro concepto computado (DISS(A,B)) que representa el conjunto de propiedades que pertenecen a A pero no pertenecen a B (ni subsumen a ninguna de las de B). La similitud entre la consulta y cada caso de la biblioteca se representa explícitamente creando los conceptos LCS(C_{new}, C_i), para todo $C_i \in$ Base de Casos. Estos conceptos se clasifican de manera automática en una jerarquía (usando la relación de subsunción) de forma que los más específicos representan las similitudes máximas con el nuevo caso. Si llamamos E_{LCS} al conjunto de conceptos más específicos de esta jerarquía, entonces para cada concepto $LCS_i \in E_{LCS}$:

- Si corresponde a un único caso (sólo tiene como instancias a C_{new} y a un único C_i) entonces se añade C_i al conjunto de casos recuperados.
- Si corresponde a varios casos entonces se les aplica el criterio DISS.

La aplicación del criterio DISS utiliza los conceptos DISS(C_1,C_{new}) y DISS (C_2,C_{new}) que son comparados mediante la relación de subsunción, de forma que se recuperan las instancias que pertenecen al concepto DISS más general, ya que es el que minimiza las diferencias.

4.2.4 Organización de la base de casos

Una vez que se ha decidido qué información se incluye en cada caso y cómo representarla, queda tratar el tema de la organización del conjunto de casos de forma que se facilite el acceso a los mismos. Éste es el problema de la indexación de la memoria de casos y conlleva dos tareas: determinar o asignar los índices de los casos y organizar dichos índices.

Hemos descrito en el apartado anterior que las funciones de similitud indican cómo comparar dos casos, pero existe un segundo ingrediente en el conocimiento de similitud referido a qué se debe comparar. De entre todos los atributos de un caso, el diseñador de un sistema CBR debe decidir cuáles se deben utilizar en el cómputo de la similitud, es decir, debe elegir un conjunto de *índices* que guíen el proceso de recuperación. Es decir, la indexación está íntimamente ligada con el proceso de recuperación y cálculo de similitud, y la organización de la base de casos debe conseguir que los casos similares se localicen eficientemente.

4.2.4.1 Asignación de índices

En las dos perspectivas presentadas (representación plana y representación estructurada) se utiliza un conjunto de atributos para describir el problema abordado por los casos. De todos esos atributos hay que decidir cuáles son (si no son todos) los que van a actuar como índices para los casos, es decir, a través de cuáles va a ser posible localizar los casos relevantes para

[3] Esta aproximación se aplica en el contexto de un sistema CBR interpretativo, donde la solución a un problema corresponde a determinar su clasificación o interpretación.

solucionar un nuevo problema. En consecuencia, los índices de los casos sirven para distinguir unos casos de otros, para predecir la utilidad de los casos. La característica fundamental de un índice es que sea *predictivo*, esto es, que permita identificar las situaciones en las que los casos pueden aportar información útil.

Podemos decir entonces que la información contenida en la descripción de un caso pertenece a uno de estos dos grupos [Watson97]: información que contribuye a la indexación, y que, por lo tanto, será útil en la recuperación, e información que no contribuye a la indexación, y que, en consecuencia, proporciona información contextual pero que no es utilizada en la localización de casos relevantes. Por ejemplo, en un sistema de diagnóstico se pueden considerar como elementos de la descripción la edad del paciente y una fotografía del mismo que muestre su estado al comienzo de la enfermedad. Ahora bien, la edad puede utilizarse como índice mientras que la fotografía puede calificarse como información no indexada, de modo que sirva para registrar el aspecto de los pacientes pero no para llevar a cabo la recuperación de los casos.

En [Kolodner&Leake96] se sugiere que para determinar los índices de los casos hay que considerar la tarea que se quiere resolver y elegir como índices los conjuntos de características que describen que un caso será útil para resolver dicha tarea. Este enfoque es radicalmente diferente a la indexación que se utiliza tradicionalmente en recuperación de información y en bases de datos. En estos ámbitos, los índices se escogen de manera que sean las características que dividen el conjunto en particiones aproximadamente similares en tamaño. Lo ideal es mantener las estructuras organizativas equilibradas, de modo que una característica es un buen discriminante (buen índice) si ayuda a mantener equilibrada la estructura. En CBR, el problema se concibe de otra manera. Se requieren índices que distingan los casos entre sí con respecto a algún objetivo. No se trata de equilibrar la estructura organizativa sino de obtener porciones conceptualmente útiles de la memoria de casos.

Los índices se pueden identificar usando la técnica de adquisición de conocimiento que consiste en entrevistar a un experto que identifica cuáles son las características críticas. Otra opción consiste en utilizar técnicas inductivas que ayudan a identificar cuáles son las características más discriminantes.

Existen distintos algoritmos clásicos para aprender, durante una etapa de entrenamiento, la importancia de las características en la valoración de la similitud, por ejemplo el algoritmo CBL4 [Aha91], basadas en explicaciones [Brown93], o razonamiento introspectivo [Leake93] [Leake *et al.* 95] [Leake95b]. Sin embargo, este tipo de algoritmos no son adecuados para sistemas en los que la importancia de una característica no es fija para todos los casos sino que puede variar en función del resto de características que describen el caso, o incluso ser distinta para distintos usuarios. En este contexto, se requieren sistemas CBR *interactivos* en los que el usuario, además de describir el problema, describe el criterio de calidad de los casos recuperados, esta aproximación se corresponde con la aproximación basada en criterios de relevancia descrita en [Ashley&Aleven93].

4.2.4.2 Estructuras de organización de la base de casos

En [Lopez&Plaza97] los autores se refieren a la organización de la memoria de casos como un problema abierto, en el que una buena indexación no es suficiente. Una vez que los índices han sido determinados, éstos pueden organizarse de manera que el proceso de localización de los casos relevantes se optimice[4]. Normalmente las mismas pautas que se utilizan para determinar la similitud se usan para construir los esquemas de organización que repre-

[4] Esta organización no es obligatoria, también puede optarse por no estructurar los índices de ninguna manera.

senten esta medida. Es decir, si en un sistema se quiere utilizar una cierta función de similitud se elegirá como la estructura de organización de la memoria más adecuada, a aquella que me permita definir sobre ella un algoritmo de recuperación eficiente y que obtenga resultados precisos, es decir, que encuentre siempre el caso más similar.

Para que un modelo de memoria sea flexible el conocimiento que contiene debe ser accesible incluso si los criterios expresados en la consulta encajan sólo parcialmente con las características representadas en el conocimiento almacenado. Esto es una necesidad ya que una nueva experiencia muy raramente satisface totalmente nuestro modelo del mundo. Por esta razón, las teorías psicológicas de la memoria se centran normalmente en un modelo basado en similitud para la representación del conocimiento y la recuperación.

Sin embargo, una opción que no depende de la medida de similitud consiste en utilizar una organización plana que no añade estructura adicional a los índices ni a los casos. Esta organización requiere un algoritmo de recuperación que consiste en el recorrido lineal de los casos lo que supone una complejidad lineal que sólo es razonable para bases de casos pequeñas. Como ventaja el algoritmo garantiza que siempre encuentra el caso más similar.

Debido a los requisitos de eficiencia del proceso de recuperación y como un medio de evitar la búsqueda exhaustiva, se puede proponer una organización de la memoria sistemática y dirigida por el tipo de recuperación que se va a hacer sobre ella. La organización jerárquica de los índices permite seleccionar un subconjunto del total de los casos y sólo se computa la similitud entre la consulta y los casos del conjunto. El problema asociado con esta aproximación es que a menudo no se puede asegurar que el que devuelvo es el mejor.

Un esquema de organización muy habitual consiste en construir un árbol de decisión, de forma que sean los valores de los índices los que permitan situar el caso en una parte u otra de la estructura. Algunas de estas estructuras jerárquicas son las redes de características compartidas y las redes discriminantes. Como ejemplo citamos el uso de árboles de decisión inducidos mediante el algoritmo ID3 o alguna de sus variantes [Kolodner93]. A partir del conjunto de casos, y una vez fijados los atributos que actúan como índices, se induce un árbol de decisión que permite recuperar en un número mínimo de pasos los casos más relevantes para una consulta dada. El problema básico de los árboles de decisión es que sólo comparan la igualdad de características y no permiten manejar adecuadamente las consultas incompletas. Por ello se han desarrollado algunos refinamientos como los árboles de decisión redundantes –uno por cada permutación de los índices.

De mayor difusión son los árboles k-d, una estructura de datos que resulta de generalizar los árboles de decisión desarrollada inicialmente en aplicaciones de informática gráfica. Esta estructura está pensada para, dado un conjunto de puntos repartidos en un espacio métrico k-dimensional, recuperar eficientemente los k vecinos más próximos (*k-nearest neighbors*) a un punto dado. No se plantea la recuperación de "el" vecino más próximo porque análisis teóricos demuestran que la complejidad de un algoritmo que recuperase exactamente el más próximo sería mucho más alta que la de simplemente recuperar los k más cercanos. El árbol k-d se induce, mediante un algoritmo sofisticado, a partir del conjunto de puntos –casos– inicial, de forma que en cada hoja del árbol haya como máximo un cierto número n de puntos. El problema geométrico consiste en dividir un espacio k-dimensional en volúmenes que contengan un número homogéneo de puntos.

La característica común de las estructuras anteriores es que se construyen como un medio de evitar la búsqueda exhaustiva durante la recuperación, pero no aportan conocimiento adicional aparte del conocimiento de similitud que se podría obtener al computar la medida de similitud que la estructura de organización explicita.

Además de las anteriores existe otro tipo de estructuras, que también organizan la base de casos pero no sólo contienen conocimiento de similitud, sino conocimiento adicional sobre las relaciones entre los elementos que describen los casos. Por ejemplo, las redes de activación (*Spreading Activation Networks*) son una adaptación de las redes neuronales donde la memoria de casos se representa como una red de nodos interconectados. La recuperación se implementa como un proceso que activa los nodos asociados con la consulta, y propaga la activación a otros nodos. Este proceso depende en gran medida de la estructura de la red, es decir, de si existe un camino entre los nodos del caso consulta y del caso recuperado. Por tanto, la red no representa una única medida de similitud entre los casos ni dicha medida es simétrica, ya que las activaciones pueden no depender únicamente del caso consulta sino que se incluyen restricciones adicionarles, inherentes a la estructura de memoria en sí mismas, que guiarán la búsqueda [Beghardt *et al.* 97]. Esta aproximación se ha utilizado también en sistemas como CREEK [Aamodt91] y CRASH [Brown93] donde se considera que el uso de índices requiere una organización de la biblioteca de casos de una forma demasiado específica del dominio o de la aplicación, ya que hay que predecir a priori las situaciones en las que un caso será recuperado. Como alternativa propone reemplazar los índices con una representación del conocimiento en forma de red y un proceso de recuperación basado en activación de nodos que es flexible y eficiente. Centra su propuesta en una explotación de una memoria muy organizada y estructurada, pero con la intención de imponer restricciones mínimas respecto a cómo se va a utilizar un determinado caso en el futuro.

También existe una línea clásica de trabajo en modelos complejos de memoria que se suelen utilizar en sistemas sofisticados, ya que el objetivo de la estructura de memoria no sólo es el de optimizar el algoritmo de recuperación sino que además incluyen conocimiento adicional en forma de generalizaciones de los casos. Dentro de esta opción, la aproximación más común consiste en tener una estructura jerárquica en la que los nodos internos son generalizaciones de los casos individuales, como en el sistema CYRUS [Kolodner83] basado en el modelo de memoria dinámica de [Schank82]. La memoria de casos en este modelo es una estructura jerárquica de *episodic memory organization packets* (MOPs). La idea básica es organizar los casos específicos que comparten propiedades similares bajo una estructura más general o episodio generalizado (GE, *generalized episode*). Un GE contiene normas, casos e índices. Las normas son características comunes a todos los casos indexados bajo un GE y los índices son características que discriminan entre los casos de un GE. La memoria de casos es una red de discriminación en la que cada nodo es un GE, un índice o un caso. Un caso se recupera encontrando el GE (nodo de la jerarquía) que tiene más normas en común con la descripción del problema y los índices (que son los nodos bajo el GE) son recorridos para encontrar el caso que tenga más de las restantes características del problema. La estructura de la memoria es dinámica porque las partes similares entre los dos casos se generalizan dinámicamente en un nuevo GE.

En el sistema PROTOS [Bareiss88] [Bareiss *et al.* 90] se utiliza una organización jerárquica alternativa para los casos. La memoria de casos queda embebida dentro de una estructura en red de categorías, relaciones semánticas, casos e índices (que se representan como punteros). Cada caso está asociado con una categoría y los índices pueden apuntar tanto a un caso como a una categoría. Existen índices de tres tipos. Los *recuerdos* conectan las características de los problemas con casos o categorías. Los *índices de casos* conectan las categorías con sus casos (ejemplares). Y los *índices diferencia* conectan los casos con otros casos que difieren en un número pequeño de características. Además, las características están entrelazadas en una red semántica que representa conocimiento terminológico del dominio (ver Apartado 4.2.2) y permite proporcionar soporte explicativo a las tareas del CBR. Para buscar un caso se combinan las características del problema de entrada en un puntero al caso o categoría que com-

parte la mayoría de las características. Si un recuerdo apunta a una categoría se recuperan sus casos prototípicos. La red semántica de conocimiento del dominio se usa para permitir el encaje entre características que son semánticamente similares.

Las estructuras conceptuales utilizadas en las aproximaciones anteriores pueden variar en su complejidad, aunque en ocasiones su representación puede suponer cierto esfuerzo de adquisición de conocimiento (aunque es un conocimiento menos profundo y más fácil de adquirir que el que define el cuello de botella de la adquisición de conocimiento).

La definición de los conceptos adecuados para organizar los casos es conocimiento adicional que se codifica manualmente utilizando técnicas de adquisición de conocimiento tradicionales de los KBSs. Otra opción con menos coste, se basa en aplicar técnicas inductivas para extraer generalizaciones de los propios casos. En esta línea de trabajo se encuentra sistemas como MMA (*Massive Memory Architecture*) [Plaza&Arcos94] e INRECA (*Induction and Reasoning from Cases*) [Bergmann *et al.* 97] [Bergmann *et al.* 99].

Las generalizaciones de casos que aparecen en las estructuras anteriores también están relacionadas con la propuesta del sistema PARIS [Bergmann&Wilke96] de representar los casos en distintos niveles de abstracción. Los casos abstractos ubicados en los distintos niveles de abstracción pueden utilizarse como índices jerárquicos para aquellos casos (concretos o abstractos) que contienen el mismo tipo de información pero a un nivel de abstracción menor. Se puede construir una jerarquía de abstracciones en la que los casos abstractos están en la parte alta de la jerarquía. Los nodos hoja contienen los casos concretos. Durante el proceso de recuperación, esta jerarquía se recorre en sentido descendente siguiendo únicamente aquellas ramas en las que los casos abstractos son suficientemente similares al problema actual. Los casos abstractos son usados como índices de los casos más concretos. Este tipo de indexación hace una suposición implícita relativa a la valoración de la similitud: requiere que un problema no pueda ser similar a un caso concreto a menos que sea similar a ese mismo caso a un nivel mayor de abstracción. Este tipo de organización de la memoria es similar a los MOPs de CYRUS [Schank82]. Igual que en el modelo de Bergmann y Wilke, se puede decir que en general, la mayoría de los sistemas de CBR usan organizaciones de memoria inspiradas en el modelo de Schank o de Porter o en alguna combinación de ellas. Otro ejemplo es el sistema BOLERO [Lopez&Plaza93] que usa los episodios generalizados de Schank junto con los enlaces a los ejemplares y los prototipos de Porter.

Como última aproximación citamos los trabajos que se basan en organizar los índices y los valores dentro de un modelo taxonómico de conocimiento terminológico del dominio [Napoli *et al.* 96] [Napoli&Lieber96] [Kamp96] [Napoli *et al.* 97] [Koelher94] [Gómez-Albarrán00] [González *et al.* 99b] que hemos comentado en el Apartado 4.2.2 y en los que se enmarca nuestra aproximación, por lo que incidiremos en ellos en los capítulos siguientes.

4.3 Recuperación

Cualquier método de recuperación combina algún procedimiento para valorar el grado de similitud entre dos casos y algún procedimiento para buscar el caso más similar a la consulta. Respecto al procedimiento de búsqueda la investigación se refiere a limitar o acotar la búsqueda para poder encontrar el mejor caso de forma más eficiente sin reducir la calidad del resultado. Respecto a la valoración de la similitud entre los casos existen dos aproximaciones básicas [Porter89] y numerosas variantes suyas:

- La aproximación *representacional* en su forma más extrema representa de forma explicita la función de similitud de forma previa a la recuperación, es decir, los casos en la base de casos residen en una estructura de datos, por ejemplo un grafo, de forma

que la proximidad en dicha estructura denota similitud. [Porter89] [Brown93] [Napoli *et al.* 97] [González *et al.* 99b] [Gómez-Albarrán00]. Esta aproximación se caracteriza por su eficiencia durante la recuperación, ya que no se computan valores de similitud y la estructura de datos está optimizada para la recuperar casos según una cierta función de similitud.

- La aproximación *computacional*, por el contrario y también en su forma más extrema, únicamente computa los valores de similitud durante la ejecución del proceso de recuperación. Se pueden utilizar ciertos índices para recuperar un conjunto de casos y después computar la similitud para cada uno de ellos de forma lineal.

Esta última aproximación, simple y muy utilizada, consiste en llevar a cabo una búsqueda exhaustiva en la base de casos comparando cada caso con la consulta. Sin embargo, en sistemas reales con un gran número de casos y una función de similitud costosa de computar, la recuperación lineal puede resultar de una eficiencia intolerable. Por ello la comunidad CBR ha propuesto el uso de arquitecturas paralelas que se apoyan en un aumento de los recursos hardware utilizados, o como hemos descrito en el apartado anterior, ha desarrollado o adaptado estructuras de organización y algoritmos de recuperación que las recorren y evitan tener que examinar todos los casos.

En esta línea, la mayoría de las aproximaciones consiguen este objetivo procesando los datos en bruto para obtener una biblioteca de casos optimizada que facilite un procedimiento de búsqueda dirigida, usando estructuras de índices más o menos complejas, obtenidas manualmente o mediante alguna técnica automática. Otras aproximaciones a la recuperación han propuesto, técnicas alternativas como los métodos de propagación de activación en una red semántica, o realizar una búsqueda exhaustiva sobre una base de casos reducida [Smyth&McKenna99].

En general, el algoritmo de recuperación dependerá directamente de la estructura de organización de la memoria de casos. Aunque no vamos a detallarlos, cada una de las estructuras de organización descritas en el Apartado 4.2.4 define un algoritmo de recuperación asociado. Para todos ellos el objetivo es obtener, a partir de la descripción de un problema, los casos que mejor se ajusten a dicha descripción, esto es, los que resuelvan un problema lo más similar posible al nuevo problema que necesitamos resolver.

La recuperación se puede subdividir en tres tareas:

1. Identificación de las características o índices que describen el nuevo problema. La nueva situación es analizada y se identifican los índices y sus valores que caracterizan el problema a resolver, utilizando el mismo vocabulario que para los casos que hay almacenados en la memoria de casos.
   En los enfoques menos sofisticados se pide al usuario que construya un vector de pares atributo-valor que represente el problema consulta. En el otro extremo se encuentran los enfoques más ricos en conocimiento que pueden realizar ciertas inferencias sobre la consulta a medida que ésta se va introduciendo. De esta manera se pueden deducir, por ejemplo, que ciertos valores de unos atributos implican restricciones sobre los valores de otros atributos.

2. Localización de los casos relevantes, que serán los que más se ajustan a dicha descripción según la estructura.
   Una vez identificados los valores de los índices del caso consulta, se utilizan para localizar los casos más relevantes. Como ya hemos destacado, se trata de una localización aproximada, en la que los casos relevantes se obtienen comparando el nuevo problema con los problemas descritos en la base de casos. Dicha comparación se

puede realizar basándose en características sintácticas superficiales o, si se dispone de un conocimiento más profundo de las características utilizadas en las descripciones, mediante algún tipo de comparación *semántica*. Como ya hemos comentado en el proceso de búsqueda se pueden usar distintas aproximaciones ligadas a los esquemas de indexación que se utilicen.

3. Selección, de entre los casos más relevantes, del mejor candidato.
   Una vez recuperados un cierto número de casos considerados relevantes, se debe seleccionar el más adecuado para el problema actual. En algunos sistemas este proceso no es un proceso independiente de la tarea de localización: la propia localización de casos relevantes devuelve los casos ordenados por similitud con el nuevo problema y el mejor candidato es el caso que ha obtenido una mayor similitud. En otros sistemas el proceso de selección puede ser más sofisticado realizándose un segundo análisis, más detallado, de los casos obtenidos inicialmente. Incluso en las aproximaciones más ricas en conocimiento se pueden encontrar sistemas que, basándose en el conocimiento general, son capaces de generar explicaciones sobre las diferencias entre el problema propuesto y los casos recuperados [Kolodner93].

4.4 Adaptación

El aspecto clave del CBR como paradigma de resolución de problemas es que las soluciones a nuevos problemas son generadas aplicando las soluciones recuperadas de problemas similares previamente resueltos, en lugar de partir de cero. En muchas ocasiones, las soluciones así obtenidas son adecuadas y se pueden aplicar directamente. En otros casos, están próximas a la solución requerida pero no lo suficiente, por lo que resulta necesario adaptarlas. Ahora bien, la adaptación es el proceso del ciclo menos estudiado y donde no existen técnicas estándar. Muchos sistemas CBR, sobre todos los del ámbito no académico, obvian la fase de adaptación o la dejan en manos de los usuarios total o parcialmente. Muchos expertos coinciden en que, más que como sistemas autónomos, los sistemas de CBR deben actuar como *asesores* de adaptación [Leake95a] [Leake *et al.* 96]. Se apuesta, por lo tanto, por dejar el proceso bajo control del usuario y que el sistema sugiera puntos de la solución que necesiten ser adaptados, e informe de restricciones e interacciones que deban ser tenidas en cuenta.

Entre los sistemas que incluyen adaptación automática se suelen utilizar técnicas *ad-hoc* que implican, en mayor o menor medida, un esfuerzo de adquisición de conocimiento que permita construir un modelo sobre los cambios aplicables a una solución y cómo determinar qué cambios aplicar a una solución en base a las diferencias detectadas entre las descripciones de los problemas. Una línea de trabajo reciente aboga por el uso de CBR en el propio proceso de adaptación. Si el sistema no tiene conocimiento suficiente para adaptar una solución, solicita la ayuda del usuario, y guarda las transformaciones que éste realiza en la solución original, lo que define un "caso de adaptación". De esta forma, cuando el sistema ha de realizar una adaptación, antes de pedir la ayuda del usuario, intenta recuperar problemas de adaptación similares y aplicarlos [Leake *et al.* 97a].

Como una manera de facilitar la tarea de adaptación, existe una línea de trabajo que se propone considerar el esfuerzo de adaptación en el proceso de selección de los casos, de manera que, al elegir el caso relevante, no sólo se tenga en cuenta lo parecidos que son los casos con el caso consulta, sino también una medida de la dificultad que entraña adaptarlos [Smyth&Keane93] [Leake *et al.* 97b] [Smyth&Keane98].

Si se decide llevar a cabo el proceso, para adaptar la solución del caso recuperado en el contexto del nuevo problema será necesario:

- Identificar las diferencias entre ambos, determinando así qué partes de la solución se pueden transferir directamente al nuevo caso y qué partes necesitan ser modificadas.
- Aplicar mecanismos que, teniendo en cuenta esas diferencias, sugieran cambios apropiados. Los cambios pueden ser *simples* consistiendo, por ejemplo, en la sustitución de una componente de la solución por otra, o complejos, como cuando es necesario modificar la estructura completa de la solución.

4.4.1 Identificando qué hay que adaptar

En [Kolodner93] se distinguen diversos mecanismos para identificar qué partes de una solución previa deben ser modificadas para ajustarse a la nueva situación. Algunos de ellos son:

- Utilización de diferencias entre las descripciones de los problemas. Este método se basa en dos procesos:
  - Encontrar las diferencias entre el problema asociado al caso recuperado y la nueva situación.
  - Tener especificadas las conexiones entre aspectos de la descripción del problema y aspectos de la solución.

  La forma de actuar consiste en evaluar las diferencias entre las descripciones de los problemas (el previo o recuperado y el actual) y usar las conexiones establecidas para identificar qué aspectos de la solución previa necesitan ser modificados. Ambos procesos colaboran, por lo tanto, en la localización de los aspectos a modificar aunque cada uno se realiza en un momento diferente del desarrollo del sistema de CBR: la obtención de las diferencias se realiza durante el uso del sistema y la especificación de las conexiones debe estar hecha de antemano
- Un caso particular del anterior ocurre cuando las soluciones y las descripciones de los problemas comparten elementos. Entonces se compara directamente la solución propuesta con la descripción de la nueva situación localizando inconsistencias.
- Utilización de una lista de comprobación. En situaciones en las que las descripciones de los problemas de los casos sean grandes –lo que hace que el cálculo de las diferencias entre problemas sea ineficiente y establecer las conexiones pueda llevar mucho tiempo– o en las que sea necesario realizar inferencias complejas para obtener las características que guíen la adaptación, el mecanismo anterior no es apropiado.

  Una lista de comprobación contiene un conjunto de pruebas que permiten identificar posibles situaciones problemáticas estándares. Cada prueba permite identificar un aspecto de una solución que requiere ser modificado; y asociado a cada prueba hay una adaptación o un conjunto de adaptaciones estándar que serán aplicadas si la prueba indica que la situación problemática está presente.

4.4.2 Tipos de adaptación

En general, hay dos tipos fundamentales de adaptación en CBR que dependen del tipo de solución almacenada en los casos [Kolodner93], [Riesbeck&Schank89] [Watson97]:

- Adaptación transformacional o estructural. Aplica operadores o fórmulas de adaptación directamente a la solución almacenada en los casos.

- Adaptación derivacional. Se vuelve a aplicar el proceso de obtención de la solución original, utilizando las nuevas características, para obtener una solución al problema actual. Para ello se deben incluir trazas de razonamiento que almacenen conocimiento sobre el proceso derivativo que condujo a la solución.

En la literatura –ver, por ejemplo, [Kolodner93] [Riesbeck&Schank89] [Smyth&Cunningham93] [Watson97] [Bergmann&Wilke98]– se describen diversas técnicas de adaptación utilizadas en sistemas de CBR. Estas técnicas no son excluyentes, de manera que en un mismo sistema se pueden implementar varias. Describimos algunas de ellas:

- Reinstanciación. Esta técnica se utiliza cuando existen atributos del problema actual cuyos valores son diferentes a los del problema previo y dichos valores forman parte de la solución. Una adaptación mediante reinstanciación conlleva realizar:
  - Una abstracción del marco del problema y de la solución previos (mediante el uso de variables para los diferentes atributos).
  - Las correspondencias entre los atributos de los problemas previo y actual.
  - Una instanciación del marco del problema y de la solución previos teniendo en cuenta las correspondencias establecidas.

  Esta técnica es empleada, por ejemplo, en el sistema de planificación CHEF [Hammond89] que sirve para generar recetas de cocina. Cuando CHEF crea una receta de pollo con guisantes a partir de una receta de ternera con brécol, utilizando reinstanciación sustituye la ternera por el pollo y el brécol por los guisantes. El papel *carne* que ocupa la ternera en la receta inicial pasa a ser ocupado por el pollo y el papel *verdura* que ocupa el brécol en la receta inicial es ocupado por los guisantes en la nueva solución[5]. Los nuevos valores asociados a los papeles se utilizan directamente para llevar a cabo la sustitución.
- Ajuste de parámetros. La técnica de ajuste de parámetros tiene sentido cuando tanto los problemas como las soluciones tienen atributos con una cierta escala de valores. Cuando, atendiendo a los parámetros o características, una nueva situación difiere en cierto grado de una situación previa, la técnica de ajuste de parámetros cambia los parámetros numéricos[6] de la solución previa mediante interpolación de valores. El ajuste de parámetros es un proceso compuesto por dos pasos. Primero, se comparan las descripciones del problema nuevo y del problema original y se extraen las diferencias. El segundo paso consiste en aplicar heurísticas de ajuste especializado a la solución original para obtener la nueva. Dichas heurísticas capturan las relaciones entre los parámetros de los problemas y los de las soluciones, y tienen el aspecto de las reglas de producción. De hecho, a menudo se implementan como tales.
- Búsqueda local. Esta técnica consiste en buscar, en una jerarquía taxonómica de conceptos del dominio –por ejemplo, conocimiento terminológico- un concepto cercano *B* a uno dado *A* que pueda servir como sustituto de este último. En su forma más simple, la búsqueda local accede a hermanos del elemento antiguo y comprueba si sirven; si no sirven, busca entre los primos y así sucesivamente. Hay que tener en cuenta que la realización de búsquedas locales no restringidas puede ser ineficiente, lo que hace necesario disponer de directrices que controlen las subidas y

[5] CHEF hace uso de jerarquías de abstracción donde se encuentran clasificados los valores, de manera que los que juegan un mismo papel funcional están próximos en la jerarquía. Así, por ejemplo, ternera y pollo estarán próximos.

[6] Cuando hablamos de parámetros numéricos nos referimos no sólo a parámetros cuyos valores son números sino también a tipos escalares y enumerados.

los descensos por la jerarquía. Diferentes sistemas de CBR cuentan con diferentes guías para la navegación por la taxonomía o para indicar hasta dónde llegar.

- Búsqueda especializada. En esta técnica se dan instrucciones sobre *cómo* buscar, en una jerarquía de conceptos del dominio, el elemento sustituto que se necesita. Para ello se utilizan heurísticas de búsqueda especializada que señalan hacia partes de la jerarquía donde es probable que se pueda encontrar lo que se busca. La búsqueda especializada se utilizó por ejemplo en el sistema SWALE [Kass90].
- Búsqueda de sustitutos en varios casos. La búsqueda local es apropiada cuando el sustituto se puede encontrar en las proximidades de aquello que hay que reemplazar. La búsqueda especializada es apropiada para localizar sustitutos en lugares concretos de la memoria. Sin embargo, otra aproximación –que se ha utilizó por ejemplo en el sistema CLAVIER [Barletta&Hennessy89]– se basa en obtener los sustitutos en otros casos, por lo que debe ser posible buscar casos que tengan partes similares a la parte de la solución de partida que necesita ser adaptada.
- Reparación guiada por modelo. Este método, ejemplificado en el sistema CASEY [Koton89], utiliza heurísticas basadas en modelos causales de un tipo de sistema o situación. Una vez evaluadas las diferencias con respecto al modelo y clasificadas por tipo, se realizan las reparaciones asociadas con cada diferencia. Las heurísticas de reparación guiada por modelos son de propósito general y se basan en nuestro conocimiento sobre causalidad. Hay una asociada con cada tipo de cambio que uno puede realizar en una explicación causal.
- Adaptación jerárquica. Esta estrategia va más allá de ser una propuesta de adaptación; afecta a las distintas tareas del ciclo CBR. Los casos se almacenan en varios niveles de abstracción y la adaptación se realiza siguiendo un enfoque descendente: primero la solución se adapta al más alto nivel de abstracción (omitiendo los detalles menos relevantes) y luego se refina paso a paso añadiendo los detalles que faltan. Ejemplos de sistemas que usan esta estrategia son PARIS [Bergmann&Wilke96] y el enfoque estratificado descrito en [Branting&Aha95].
- La aproximación utilizada en el sistema RESYN/CBR y descrita en [Lieber&Napoli98] se basa en la determinación de los *caminos de similitud* entre el caso consulta y el caso recuperado sobre la jerarquía de abstracción, y en basar el proceso de adaptación en dicho camino de similitud. Dado un problema objetivo (*target)* se buscará en la jerarquía un caso (*source*) cuyo índice (*idx*) subsuma al del problema objetivo. Este proceso devuelve un *camino de similitud*: $SIM(source, target) = Source \subseteq idx(source) \supseteq idx(target)$. El proceso de adaptación utiliza este camino para construir una solución para el problema objetivo (basada en las diferencias con el problema fuente). Utiliza operaciones de *generalización y especialización*. El primer paso de adaptación corresponde a la relación $Source \subseteq idx(source)$ y consiste en construir una *generalización* de Sol(*source*) resultando en Sol(idx(*source*)). El segundo paso de adaptación corresponde a la relación $idx(source) \supseteq idx(target)$ y consiste en la construcción de una *especialización* de Sol(idx(*source*)) resultando en Sol(idx(*target*)). Cuando falla la *clasificación anterior* se lleva a cabo la *clasificación aproximada*, cuyo objetivo es encontrar el caso que pueda ser más fácilmente transformado, es decir, el caso para el que el coste de adaptación de su solución a la nueva situación es el menor.
- La adaptación basada en casos, que ya hemos comentado, es una línea de trabajo reciente que propone el uso de CBR para resolver el propio proceso de adaptación de

casos. Esta aproximación, ejemplificada por el sistema DIAL [Leake *et al.* 96] [Leake *et al.* 97a], supone un modelo de adaptación con un nivel alto de intervención del usuario, es decir, es adecuado para sistemas manejados por usuarios expertos. Así, si el sistema no tiene conocimiento suficiente para adaptar una solución, solicita la ayuda del usuario que realiza la adaptación manualmente. El sistema *observa* las modificaciones y en base a ellas define un "caso de adaptación". De esta forma, en procesos de adaptación futuros, antes de pedir la ayuda del usuario, intenta recuperar casos de adaptación anteriores y aplicarlos.

4.5 Revisión

Sólo una minoría de los sistemas de CBR implementan el proceso de revisión. Si la solución generada en las fases anteriores no es correcta, aún así se puede "aprender de los errores" y repararla. Es decir, la reparación de una solución es una adaptación llevada a cabo cuando se sabe que la solución no es válida. La revisión consta de una fase de evaluación de la solución seguida de la reparación de los errores.

Por lo tanto, en primer lugar, hay que tener capacidad para evaluar la corrección de las soluciones, ya sea porque la solución se pueda probar de alguna forma, por ejemplo aplicándola en el sistema real o en algún tipo de modelo, o porque un agente externo al sistema (el usuario) la señale como correcta o errónea.

Una vez evaluada la solución, si ésta no es válida, se pasa a la fase de reparación. La evaluación de la solución normalmente hace que se disponga de más información sobre la razón por la que no funciona de la que se disponía para llevar a cabo la adaptación, porque en otro caso no se generaría una solución errónea. Por ello se puede llevar a cabo la reparación utilizando más, y mejores, heurísticas y métodos, que son aplicables en la fase de revisión pero no lo eran en la de adaptación. El primer paso dentro de la reparación suele consistir en la identificación de los errores para en la modificación de la solución de modo que los fallos no vuelvan a ocurrir.

4.6 Aprendizaje

Según la definición de D. Aha [Aha97], un algoritmo de aprendizaje en cualquier contexto es un algoritmo que lleva a una mejora del rendimiento a lo largo del tiempo. El término aprendizaje en IA es a menudo sinónimo de generalización, principalmente a través de técnicas inductivas. El uso de técnicas de IA para aprender conocimiento de indexación y de adaptación a partir de la biblioteca de casos se basa en el contenido de la base de casos y no requiere una comprensión previa del dominio.

Además de un paradigma de resolución de problemas, el CBR es un paradigma de aprendizaje máquina. Por un lado induciendo generalizaciones basadas en las similitudes detectadas entre los casos y principalmente acumulando casos que serán utilizados en procesos de resolución posteriores. De esta forma, con el tiempo un sistema CBR mejora su capacidad de resolución de problemas.

El primer problema que debe tratar el proceso de aprendizaje es decidir qué casos se aprenden. La eficiencia de un sistema CBR se puede degradar cuando el número de casos crece excesivamente y, por lo tanto, se debe evitar incluir casos que no aporten información nueva al sistema. Además, en algunas aproximaciones se incluyen mecanismos de olvido de casos no útiles para el sistema. El rango de posibilidades va desde los sistemas que, estudian-

do la base de casos de forma autónoma, deciden qué casos incluir hasta los que delegan esta responsabilidad en el usuario.

La segunda cuestión relacionada con el aprendizaje es la que se refiere a la organización de la estructura de casos. Dependiendo de la complejidad de la estructura utilizada, este proceso puede ser más o menos sofisticado. Por ejemplo, si la organización es lineal bastará con añadir un nuevo elemento a la lista; si la estructura se induce a partir de los casos, será necesario recomputar periódicamente la estructura de indexación; y en los modelos más complejos donde se representan generalizaciones de los casos, es necesario aplicar técnicas de aprendizaje más sofisticadas.

Los trabajos acerca del mantenimiento de la base de casos incluyen técnicas para identificar casos poco utilizados y que por lo tanto puede interesar eliminar u olvidar, así como casos defectuosos o redundantes que perjudican a la efectividad del ciclo CBR. En concreto, estas técnicas tratan de solucionar el *problema de la utilidad* [Smyth&Keane95] que ocurre cuando el coste de buscar el conocimiento es mayor que el beneficio obtenido al utilizarlo.

Como paradigma de aprendizaje, el CBR se enmarca dentro del aprendizaje perezoso (*lazy learning*), para el que se han utilizado nombres como algoritmos basados en memoria, en instancias, en ejemplares, en casos o en la experiencia [Aha97]. La principal característica de los algoritmos de aprendizaje perezoso puros es que retrasan el procesamiento hasta que hay una petición de información, de forma que almacenan los datos de entrada y responden a una petición de información (o resuelven un problema) combinando el conocimiento de almacenado en datos almacenados (datos de entrenamiento).

En contraposición los algoritmos "impacientes" (*eager*) compilan los datos de entrada para producir descripciones intensionales de los conceptos representadas mediante reglas, árboles de decisión o redes neuronales. En este grupo se encuentran los algoritmos inductivos que descartan los datos de entrada y generan las respuestas a una petición de información utilizando la descripción inducida a priori.

CBR como paradigma de aprendizaje tiene muchas ventajas:

- Proporciona mejoras de rendimiento: razonamiento más rápido que desde cero, capacidad de anticipar y evitar errores pasados, capacidad de centrarse primero en las partes más importantes de un problema.
- El aprendizaje es simple ya que no requiere una comprensión profunda del dominio.
- Los casos individuales o sus generalizaciones pueden servir como explicaciones que son triviales de generar.
- CBR es escalable. El mayor cuello de botella está en la elección de los mejores casos para razonar. Esto es potencialmente un problema de búsqueda masiva que se maneja gracias a la indexación o al uso de implementaciones paralelas.
- El cuello de botella de la adquisición de conocimiento es mucho menor para CBR que para otros métodos de aprendizaje, que en general, necesitan disponer de mucha cantidad de conocimiento antes de que los procesos de aprendizaje sean útiles.

5. Herramientas y aplicaciones CBR

El CBR se ha utilizado de manera satisfactoria en muchas aplicaciones tanto industriales como académicas. En la situación actual el reto consiste en formalizar el tipo de razonamiento de estas aplicaciones, para ser capaces de definir metodologías y herramientas software

que asistan durante el análisis, diseño e implementación de nuevas aplicaciones CBR. En esta línea se encuentra el trabajo desarrollado en esta tesis.

En los últimos años la comunidad de CBR ha hecho hincapié en la carencia de una metodología completa de diseño e implementación de sistemas CBR, que disponga de un modelo explícito del proceso de razonamiento, y está haciendo esfuerzos en esta línea. Como primer paso hacia una metodología CBR, en CBR-PEB (*CBR-Product Experience Base*)[7] se almacenan casos de experiencia en el desarrollo de sistemas CBR, y se define un vocabulario de naturaleza descriptiva para representar estos casos de experiencia, que es adecuado para descripciones superficiales pero resulta limitado para descripciones completas y complejas de sistemas CBR. Por tanto, un segundo paso hacía una metodología CBR consiste en desarrollar un vocabulario más rico que nos permita describir diseños de sistemas CBR. El desarrollo de una ontología terminológica como CBROnto se encuentra, modestamente en el estado actual, en esta línea de trabajo en la que también enmarcamos trabajos como la arquitectura ABC [Plaza&Arcos00] o el marco conceptual para describir sistemas CBR basado en contenedores de conocimiento –*knowledge containers* [Richter95]. Estamos de acuerdo con la opinión expresada en [Plaza&Arcos00] según la cual una metodología real para el desarrollo de sistemas CBR necesita aportaciones tanto de un sector de la comunidad más teórico que desarrolle, refine y verifique un lenguaje y vocabulario descriptivo, como de un sector más empírico que aplique los resultados teóricos a sistemas reales y realimente los conceptos teóricos tanto para aceptarlos, mejorarlos o rechazarlos.

Actualmente existen varias herramientas comerciales y académicas que facilitan el desarrollo de aplicaciones CBR; cada herramienta proporciona ciertas funcionalidades específicas y será adecuada o no, según los requisitos de la aplicación CBR a desarrollar. Estas herramientas –entornos integrados para desarrollar aplicaciones CBR– o *shells* CBR, dan soporte a los procesos CBR incorporando mecanismos típicos para representar, indexar, recuperar, adaptar, revisar y aprender casos. El objetivo de los *shells* CBR es, por tanto, proveer de mecanismos generales que puedan aplicarse a los procesos concretos que las aplicaciones CBR usan. En el caso ideal, un entorno de desarrollo de aplicaciones CBR dará soporte al ciclo CBR completo, aunque es frecuente que entornos comerciales proporcionen únicamente mecanismos de indexación, por ejemplo basados en árboles de decisión, de similitud, como métricas basadas en distancia, y algún mecanismo simple de adaptación y aprendizaje. Sólo algunas herramientas ofrecen mecanismos de representación de conocimiento general del dominio y procesos complejos que lo utilicen. De lo anterior se deduce que no todas las herramientas de desarrollo son adecuadas para todos los tipos de aplicaciones CBR y la elección de la más adecuada depende principalmente de la complejidad requerida para los casos y del número de casos que se puede manejar así como del conocimiento adicional disponible sobre el dominio y su uso esperado por parte de los procesos.

5.1 Herramientas CBR

Este apartado revisa brevemente algunas herramientas CBR. El lector interesado puede consultar [Althoff *et al.* 95] y [Watson97] donde se ofrecen sendos resúmenes comparativos de éstas y otras herramientas CBR.

Entre las herramientas comerciales más populares se encuentra **CBR-Works** (previamente S3-Case), de la empresa tecInno que se diseñó como un entorno de desarrollo de aplicaciones CBR. Presenta interfaces de usuario de fácil manejo y utiliza una aproximación orien-

[7] http://www.iese.fhg.de/Competences/QPE/QE/CBR-PEB.html (K-D Althoff y M. Richter)

tada a objetos para la representación de conocimiento terminológico sobre el dominio y de los casos. CBR-Works permite representar casos estructurados, ofrece medidas de similitud predefinidas e incluye editores que permiten definir de manera intuitiva, medidas de similitud estructuradas. Ofrece mecanismos de adaptación simples basados en la definición manual de reglas de adaptación del tipo "*si condición entonces acciones*" [Bergmann&Wilke98]. CBR-Works se ha utilizado para el desarrollo de distintos tipos de aplicaciones CBR, principalmente de comercio electrónico. Da soporte al desarrollo de sistemas de clasificación y gestiona eficientemente las bases de casos grandes aunque usa un algoritmo de recuperación exhaustiva.

K-commerce (previamente CBR3, CBR Express y CasePoint) pertenece a la empresa *Inference Corporation*. Es una familia de productos con gran éxito que domina el sector de las aplicaciones de soporte para clientes. No soporta representación estructurada de casos, aunque permite asociar información adicional a los casos, por ejemplo, sonidos, gráficos y vídeos. No gestiona bien casos con muchos valores desconocidos y no permite representar conocimiento general sobre el dominio. El sistema proporciona procesos de recuperación muy rápidos (*nearest neighbor* con estructuras de indexación) y permite manejar grandes cantidades de casos de manera eficiente. Existe un modo de mantenimiento en el que se permite modificar la medida de similitud (aunque con poca expresividad) y ajustar los vectores de pesos para expresar preferencias. El sistema emplea mecanismos de aprendizaje para incorporar casos aunque no emplea técnicas de adaptación ni revisión.

El motor CBR de la herramienta **KATE** de la empresa Kaidara (antes AcknoSoft), usa recuperación mediante un algoritmo *nearest neighbor* en el que la medida de similitud y los pesos relativos de los descriptores pueden personalizarse para ajustarse a las necesidades de cada aplicación específica. KATE hace un uso combinado de *nearest neigbour* con un proceso de inducción dinámica, en el que para cada consulta se generan preguntas que se plantean al usuario, que permiten discriminar entre los casos de la base de casos. Además, su módulo de *data mining* extrae conocimiento que está oculto en los datos, construyendo automáticamente árboles de decisión a partir de los casos.

ReMind de Cognitive Systems Inc., está disponible en dos formatos. Como una biblioteca C para ser incluida en otras aplicaciones y como un entorno completo de desarrollo. ReMind no soporta representación estructurada de casos y usa un proceso de inducción que no trabaja correctamente con casos que tengan muchos valores nulos. En cuanto a la recuperación, una cualidad de ReMind es que ofrece métodos distintos que son alternativos. Por ejemplo, algoritmos *nearest neighbor*, dos tipos de inducción –simple y guiada por conocimiento– para construir árboles de decisión y recuperación basada en plantillas (consultas tradicionales a bases de datos). ReMind soporta adaptación usando fórmulas y permite aprendizaje de casos. Utiliza conocimiento general adquirido de expertos para mejorar los procesos de indexación, recuperación, valoración de la similitud entre casos y adaptación. Además ReMind es capaz de explicar por qué los casos han sido recuperados.

ReCall de Isoft es una herramienta, codificada en C++, que ofrece una combinación de recuperación con *nearest neighbor* e inducción. ReCall utiliza representación de casos orientada a objetos que permite la representación de conocimiento estructurado sobre el dominio, casos incompletos y conocimiento incierto. Permite intercambio de información con aplicaciones externas, en particular con bases de datos. ReCall proporciona mecanismos de indexación jerárquicos que son utilizados para organizar la base de casos y para recuperar casos de manera eficiente. Incluye distintos métodos para analizar, organizar y seleccionar índices en una base de casos de manera automática basándose en técnicas inductivas. El software de ReCall incluye editores gráficos especializados para definir objetos, relaciones entre ellos, taxonomías y reglas de adaptación. ReCall soporta un mecanismo de adaptación por defecto

basado en votaciones y el uso de reglas de adaptación definidas por el usuario. Además se pueden usar llamadas externas a funciones (C++) para obtener procesos más complejos.

ESTEEM de Esteem Software Inc., soporta aplicaciones que acceden a múltiples bases de casos, una representación compleja de casos mediante estructuras anidadas (un caso puede ser parte de otro caso), que permiten asociar información multimedia. Respecto a la recuperación, ESTEEM soporta varios métodos estándar de valoración de similitud (usando pesos en las características o generación de vectores de pesos usando ID3). Además, los usuarios pueden añadir funciones de similitud propias. Durante el proceso de adaptación se utiliza un mecanismo para crear y manejar reglas de adaptación, relativamente complejas. Se ha utilizado en el desarrollo de aplicaciones de atención a clientes y diagnóstico entre otras, aunque su bajo rendimiento y su interfaz poco flexible la convierten en una herramienta poco adecuada para su uso en aplicaciones grandes.

Entre las herramientas de ámbito académico destacamos el software **Java CBR Shell** desarrollado en el AIAI (A*rtificial Intelligence Application Institute*) de la Universidad de Edimburgo. Es una herramienta especialmente adecuada para el desarrollo de aplicaciones CBR que ofrece algoritmos de *matching* basados en lógica difusa (*fuzzy*), algoritmos genéticos para aprendizaje de pesos, recuperación basada en el algoritmo *nearest neighbor* y k *nearest neighbor* adaptativo, medidas de confianza, algoritmos de diagnóstico múltiple y un número ilimitado de atributos (se ha probado hasta 600 atributos).

Otro enfoque en cuanto a herramientas CBR, se basa en disponer de código directamente modificable para desarrollar una aplicación CBR. Como ejemplo de este enfoque citamos **CBR*Tools** una biblioteca software escrita en Java que proporciona un framework reutilizable para desarrollar aplicaciones CBR. Permite reutilizar diseños e implementaciones y proporciona componentes y colaboraciones abiertas que pueden extenderse y particularizarse para satisfacer los requisitos de la aplicación CBR en desarrollo. El framework está formado por un conjunto de clases abstractas y define el modo en el que los objetos colaboran.

6. Resumen y conclusiones del capítulo

Basándose en la suposición de que las experiencias son regulares y recurrentes, el razonamiento basado en casos resuelve problemas recordando experiencias previas y generando las soluciones a los nuevos problemas a partir de las soluciones de las experiencias almacenadas. La característica común de todos los sistemas CBR es el hecho de utilizar, como punto de partida para resolver el nuevo problema, una experiencia humana similar. Es decir, los sistemas CBR llevan a cabo recuperaciones aproximadas y, de entre las experiencias recuperadas, seleccionan la mejor en base a la semejanza con el nuevo problema.

En este capítulo hemos descrito los conceptos fundamentales de los sistemas CBR, poniendo énfasis en el conocimiento que está involucrado en los distintos procesos CBR. Aunque hemos visto que existen distintos tipos de conocimiento que los sistemas CBR pueden utilizar como conocimiento de soporte para llevar a cabo sus procesos de razonamiento, en nuestro trabajo estamos interesados en los sistemas KI-CBR, que se basan en conocimiento terminológico que representa el vocabulario del dominio, y en el impacto de este tipo de conocimiento en los procesos CBR.

En particular, el conocimiento terminológico se utiliza: (i) como vocabulario para definir los casos, (ii) como una posible estructura de organización de los casos que incluye conocimiento sobre similitud, (iii) como conocimiento para la generación de explicaciones y la justificación del razonamiento, y (iv) como conocimiento de adaptación sobre el que se pueden llevar a cabo procesos de búsqueda.

Capítulo 3
ADQUISICIÓN Y REUTILIZACIÓN DEL CONOCIMIENTO

1. Introducción

Existe un interés creciente en la reutilización de software, de datos y de *conocimiento* en general, ya que esto reduce los recursos y el tiempo de desarrollo invertido en los proyectos. En particular, el alto coste de la adquisición de conocimiento hace que la reutilización sea algo esencial para los sistemas basados en conocimiento (KBSs). En 1991, desde el ARPA *Knowledge Sharing Effort* se propuso una nueva forma de construir sistemas inteligentes:

> *"Actualmente el desarrollo de sistemas basados en conocimiento supone, en general, la construcción desde cero de bases de conocimiento nuevas. Esto podría hacerse ensamblando componentes reutilizables y así, los desarrolladores de los sistemas sólo necesitarían preocuparse de la creación de conocimiento y mecanismos de razonamiento especializados, que sean nuevos para la tarea o sistema específicos. Este nuevo sistema interaccionaría con otros sistemas existentes, utilizándolos para llevar a cabo parte de sus razonamientos. De este modo, el conocimiento declarativo, los métodos de resolución de problemas y los servicios de razonamiento podrían ser elementos compartidos entre los sistemas. Esta aproximación facilitaría la construcción con un menor coste de sistemas mayores y mejores".* Traducido de *[Neches et al. 91]*.

Desde entonces y en el camino hacia el objetivo final, aún lejano, de compartir y reutilizar conocimiento y mecanismos de razonamiento a través de dominios y tareas, se han desarrollado numerosas metodologías, bases conceptuales, ontologías y métodos genéricos de resolución de problemas (PSMs, del ingles *Problem Solving Methods*).

Actualmente, la reutilización y compartición de conocimiento es un área de interés y representa un reto para la investigación en Inteligencia Artificial (IA). De hecho, una idea subyacente compartida por las principales metodologías de soporte para el diseño de KBSs es la reutilización de componentes, principalmente conocimiento del dominio en forma de onto-

logías y de resolución de problemas. Las ontologías y los PSMs pueden considerarse componentes reutilizables complementarios para construir KBSs [Gómez-Pérez&Benjamins99].

Las ontologías capturan el conocimiento de un dominio de modo genérico y proporcionan una comprensión comúnmente aceptada que puede ser reutilizada y compartida a través de aplicaciones y grupos [Chandrasekaran *et al.* 99]. Proporcionan un vocabulario común de un área y definen, usando distintos niveles de formalidad, el significado de los términos y de las relaciones entre ellos.

Los PSMs describen los procesos de razonamiento de los KBSs de forma independiente de la implementación y del dominio concreto de aplicación. En general, describen un modo de conseguir los objetivos de una cierta tarea.

El tema anterior ha puesto de manifiesto el problema del cuello de botella que supone la adquisición de conocimiento en sistemas basados en modelos explícitos de conocimiento sobre un dominio. Para el caso concreto de los sistemas KI-CBR nuestra apuesta en esta tesis es la de reutilizar conocimiento ontológico para adquirir conocimiento terminológico sobre el dominio de aplicación. Nuestro trabajo estudia si el tipo de conocimiento que encontramos en una ontología de un dominio es o no adecuado para llevar a cabo procesos CBR sofisticados que usan activamente este conocimiento, además del de los propios casos.

Este capítulo comienza con una introducción breve de algunas metodologías de desarrollo de KBSs que han influido en nuestro trabajo. El resto del capítulo se encarga de describir algunos aspectos generales relacionados con las ontologías (Apartado 3) y los PSMs (Apartado 4) –qué son, sus fundamentos, tipos, aplicaciones y técnicas– así como los aspectos relacionados con su reutilización para adquirir el conocimiento necesario en los KBSs. Para poder utilizar las ontologías y los PSMs deben estar especificados en algún lenguaje. Por ello, de entre las distintas técnicas de representación de conocimiento, en el Apartado 5 describimos la opción elegida en nuestro trabajo: las *lógicas descriptivas* (DLs), un formalismo basado en lógica que adopta un enfoque de representación "centrado en los objetos" y que proporciona potentes mecanismos para razonar con el conocimiento representado.

2. Metodologías de desarrollo de sistemas basados en conocimiento

Durante las décadas de los 80 y 90 surgieron metodologías para la Ingeniería del Conocimiento que dan soporte al proceso de desarrollo de KBSs. Sirvan de ejemplo *Generic Tasks* [Chandrasekaran86][Chandrasekaran87], PROTÉGÉ [Musen89], PROTÉGÉ-II [Puerta *et al.* 92], *Components of Expertise* [Steels90], KADS [Wielinga *et al.* 92] y CommonKADS [Schreiber *et al.* 94] [Schreiber *et al.* 00]. En este apartado resumimos las ideas básicas de algunas de estas metodologías que han influido en nuestra visión del desarrollo de sistemas KI-CBR como un proceso de reutilización de componentes, siendo los PSMs y las ontologías los candidatos principales para ser reutilizados.

Generic Tasks [Chandrasekaran86][Chandrasekaran87]es una metodología de desarrollo de KBSs orientada a las tareas, es decir, basada en el proceso de analizar y representar una estructura de tareas común para la resolución de un problema dado y especificar los requisitos del dominio asociados con las tareas de dicha estructura. Cada tarea se describe en términos de los métodos que la resuelven y de las condiciones en las que cada método es aplicable. Cada método se describe en base al uso que hace del conocimiento y la división en subtareas que lleva a cabo para resolver una tarea. El proceso de descomposición de tareas en subta-

reas se aplica recursivamente hasta que un método es capaz de resolver una tarea sin descomponerla. La metodología se basa en una biblioteca de tareas y métodos genéricos que facilita el desarrollo de nuevos KBSs.

La metodología *Components of Expertise* [Steels90] propone un marco para describir la experiencia al *nivel de conocimiento* [Newell82]. Una descripción de un sistema a dicho nivel se basa en el conocimiento que contiene y no en las estructuras de implementación de ese conocimiento. Esta metodología descompone la experiencia en tres perspectivas:

- La perspectiva de tareas especifica cuál es el conjunto de tareas a resolver. Típicamente existe una tarea principal, que determina el objetivo de la aplicación, y que se descompone en otras subtareas de forma recursiva hasta alcanzar tareas elementales. Esta descomposición forma lo que se conoce como estructura de tareas.
- La perspectiva de modelos se centra en el conocimiento disponible que es necesario para resolver las tareas. Distingue entre varios tipos de modelos, entre los que se destacan los modelos de casos y los modelos del dominio. Los modelos de casos describen situaciones específicas pendientes de resolución (tareas). Para describirlos se pueden utilizar modelos de distinto tipo, dependiendo el tipo de tarea a resolver, por ejemplo, modelos de componentes, descriptivos, de comportamiento, funcionales, temporales o causales entre otros. Después de identificar los distintos modelos de casos del sistema, el siguiente paso consiste en determinar la relación existente entre ellos. Los modelos del dominio describen conocimiento específico del dominio que los métodos utilizan para construir y modificar los modelos de casos. En esta metodología, la resolución de problemas se considera una actividad de modelado de una situación en la que se ha resuelto un problema, es decir, no hay tareas pendientes de resolución.
- La perspectiva de métodos especifica cómo y cuándo se aplica el conocimiento. Los métodos imponen estructuras de control sobre las tareas y se dividen en tres tipos: métodos de descomposición de tareas, métodos de ejecución de tareas y métodos de búsqueda de información.

CommonKADs [Schreiber *et al.* 94] [Schreiber *et al.* 00] es una metodología para el desarrollo de KBSs que surge como evolución de la metodología KADS [Wielinga *et al.* 92]. En ambas se considera el desarrollo de un KBS como un proceso de construcción de un modelo de comportamiento de resolución de problemas en un contexto concreto. Utiliza distintos modelos para expresar distintos puntos de vista de una situación de resolución de un problema, por ejemplo, modelos de tareas, de experiencia, de organización, de comunicación, de agentes y de comunicaciones. Los modelos de organización, tareas, agentes y comunicaciones capturan el contexto en el que se lleva a cabo la actividad de resolución del problema, mientras que el modelo de experiencia captura el conocimiento y los procesos de razonamiento involucrados. El modelo de experiencia se estructura a su vez en tres modelos de conocimiento –dominio, inferencias y tareas– y los mecanismos de integración entre ellos:

- El conocimiento de tareas se describe mediante una descomposición de las tareas de alto nivel en subtareas, incluyendo el flujo de control entre ellas. La descripción de una tarea incluye una componente de definición que especifica el objetivo de la tarea en términos de sus entradas y salidas, que hacen referencia al conocimiento del dominio. Este conocimiento del dominio especifica la estructura y el contenido del conocimiento específico que es relevante para una aplicación.

- El conocimiento del dominio se estructura en torno a diferentes ontologías que proporcionan puntos de vista parciales del mismo. Cada uno de estos puntos de vista, es decir, un subconjunto del conocimiento del dominio descrito por una ontología, se llama modelo del dominio. De esta forma el conocimiento del dominio puede estructurarse en varios modelos del dominio.
- El conocimiento de inferencias especifica cuáles son los pasos primitivos de razonamiento en una aplicación, y el tipo de conocimiento del dominio que participa en esos pasos. Además, la estructura de inferencias describe las dependencias de datos entre los pasos de razonamiento.

Una idea de CommonKADs, que también comparten otras metodologías, por ejemplo PROTÉGÉ-II [Puerta *et al.* 92], es la reutilización de los componentes de conocimiento de los modelos. Como parte de la metodología se han desarrollado bibliotecas de componentes reutilizables relativos a todos los aspectos que se deben modelar en un nuevo sistema usando la metodología. PROTÉGÉ-II proporciona un entorno para desarrollar KBSs que se basa en seleccionar y modificar PSMs y ontologías reutilizables. El entorno proporciona un generador de herramientas de adquisición de conocimiento basado en ontologías, que genera interfaces gráficas de adquisición del conocimiento descrito por una ontología dada.

La reutilización en PROTÉGÉ-II se basa en la existencia de una biblioteca de PSMs que resuelven tareas. Los PSMs tienen requisitos de entrada y salida y pueden descomponer una tarea en subtareas o resolverla. Los PSMs que resuelven tareas se llaman *mecanismos*. Las ontologías se definen como jerarquías de clases y se dividen en tres tipos: ontologías del dominio, ontologías de métodos y ontologías de aplicación. Las ontologías del dominio se encargan de modelar los conceptos y las relaciones de un cierto dominio. Las ontologías de métodos modelan conceptos relacionados con los PSMs, incluyendo los requisitos de entrada y salida. Para promover su reutilización las ontologías de métodos deben ser independientes del dominio de aplicación. Por último, las ontologías de aplicación combinan el dominio y los métodos para configurar una aplicación.

Uno de los objetivos fundamentales de PROTÉGÉ-II es la reutilización de PSMs y ontologías, por lo que proporciona un modo de conectar estos componentes en una aplicación usando relaciones de conexión (*mappings*) [Gennari *et al.* 94]. Además, proporcionan una ontología de conexión (*mapping ontology*) que define un lenguaje para especificar distintos tipos de relaciones de conexión entre los elementos.

Como veremos posteriormente en los capítulos de esta memoria, nuestra visión del desarrollo de sistemas KI-CBR comparte algunas de las ideas expuestas en este apartado. En particular la visión del desarrollo de los KBSs como un proceso de reutilización de componentes de conocimiento, tanto conocimiento del dominio como de resolución de problemas, la integración –conexión– y configuración de dichos componentes, la división del conocimiento de un KBS en varios *modelos* según el papel que juegan y la resolución de problemas como un proceso de descomposición de tareas en subtareas utilizando métodos que hacen uso del conocimiento del KBS.

3. Ontologías

En el Capítulo 2 se ha descrito un problema clásico asociado con los KBSs, el cuello de botella de la adquisición del conocimiento, que surge con la necesidad de adquirir el conocimiento necesario sobre el dominio sobre el que vamos a trabajar y representarlo explícitamente en un formato procesable con el que se pueda razonar de manera automática. Uno de los obje-

tivos de nuestro trabajo en esta tesis es el de reducir el esfuerzo de adquisición de conocimiento asociado a los sistemas KI-CBR. Para ello proponemos adquirir conocimiento terminológico del dominio de aplicación a partir del conocimiento previamente conceptualizado y formalizado en ontologías existentes. Este conocimiento servirá como estructura inicial sobre la que organizar otro tipo de conocimiento que es específico de la aplicación, en forma de casos concretos, con los que también se razona.

En el apartado anterior hemos visto que las principales metodologías que dan soporte al proceso de desarrollo de KBSs comparten la característica principal de promover la reutilización de conocimiento proporcionando y utilizando bibliotecas de componentes. También hemos visto que en estas metodologías los candidatos principales para ser reutilizados son los PSMs y las ontologías, de las que nos ocupamos en este apartado. Parece que existe un acuerdo general respecto a que el acceso a bibliotecas de componentes reutilizables facilitaría en gran medida el proceso de Ingeniería del Conocimiento, y muchos grupos de investigación se han ocupado de desarrollar componentes candidatos a ser reutilizados. Sin embargo, existe un debate desde hace un tiempo en el campo de la Inteligencia Artificial relativo a si es posible o no representar el conocimiento independientemente de cómo se utilice para razonar. Esto se conoce como el *problema de interacción* [Bylander&Chandrasekaran88]:

> *La representación del conocimiento con el propósito de resolver algún problema está muy influido por la naturaleza del problema y por la estrategia de inferencia que se aplique para resolver el problema.*

En opinión de [Van Heijst *et al.* 97], teniendo en cuenta el problema de interacción, es difícil que un método pueda utilizar conocimiento que está definido teniendo en mente otro método de resolución. Sin embargo, existen otros trabajos que defienden la independencia entre el conocimiento del dominio y el conocimiento de resolución de problemas. De hecho la idea de que la separación entre el conocimiento de control y el conocimiento del dominio incrementaría la reutilización de ambos tipos de conocimiento, fue una de las bases de la concepción del modelo de cuatro capas de KADS [Wielinga&Breuker86]. También N. Guarino [Guarino97], en respuesta a [Van Heijst *et al.* 97], defiende la tesis de que existe una cierta independencia entre el conocimiento del dominio y el conocimiento de resolución de problemas y la viabilidad de que una misma ontología sobre un dominio sea compartida por varias aplicaciones.

Nuestra postura en esta tesis, que se reduce únicamente al ámbito de resolución de problemas mediante CBR, comparte la idea de que el conocimiento general sobre el dominio proporcionado por las ontologías puede ser reutilizado en distintas aplicaciones y que el conocimiento de resolución de problemas en aplicaciones CBR puede ser reutilizado para distintos dominios.

Bylander y Chandrasekaran identificaron dos razones subyacentes al problema de la interacción. Primero, la tarea que resolverá la aplicación determina en gran medida el tipo de conocimiento que se debería codificar ya que, en general, no es ni necesario ni deseable modelar todo lo que el experto sabe sino sólo le que resulte útil para la resolución de dicha tarea. En segundo lugar, el conocimiento debe codificarse de un modo que la estrategia de inferencia que se utilice pueda razonar eficientemente con él.

En nuestra aplicación concreta, respecto al punto primero hemos experimentado que el tipo de conocimiento que encontramos en las ontologías de un dominio de aplicación sobre el que se quiere desarrollar un sistema CBR es adecuado para ciertos métodos que resuelven las tareas que típicamente se llevan a cabo durante el proceso de resolución de problemas con CBR. Lógicamente, este tipo de conocimiento no es el único que puede ser adecuado y existen otros métodos que se benefician de otros tipos de conocimiento que típicamente no se incluyen en las ontologías, por ejemplo, modelos exhaustivos de comportamiento o mode-

los causales, pero cuyo coste de adquisición es muy elevado. Incluso existen métodos que no utilizan ningún tipo de conocimiento adicional, aparte del de los propios casos.

En nuestra aproximación nos centraremos en el primer tipo de métodos y utilizamos las ontologías como fuente de adquisición de conocimiento para desarrollar sistemas KI-CBR, reutilizando conocimiento que ha sido previamente conceptualizado y formalizado, permitiendo beneficiarnos de su corrección, completitud y consistencia. Este apartado intenta dar una visión general sobre qué son las ontologías y cuáles son sus componentes y sus principios básicos de diseño. Dicha visión permite entender cómo podemos reutilizarlas y qué ventajas proporciona este tipo de adquisición de conocimiento en el contexto concreto de los sistemas CBR.

El Apartado 3.1 resume distintas definiciones del término Ontología existentes en la literatura y algunos aspectos relacionados con su construcción. El Apartado 3.2 incide en aspectos relativos a la reutilización de ontologías y el Apartado 3.3 resume brevemente algunas herramientas que ayudan en este proceso de reutilización.

3.1 ¿Qué es una ontología?

El término ontología ha ganado una gran popularidad dentro de la comunidad de Ingeniería del Conocimiento, especialmente debido a la iniciativa ARPA de compartición de conocimiento ([Neches *et al.* 91] [Patil *et al.* 92] [Mars92] [Gruber93] [Guarino95]). Sin embargo su uso diversificado hace que su significado sea objeto de distintas interpretaciones.

La palabra *Ontología* ha sido tomada de la Filosofía, donde significa "una explicación sistemática de lo que existe". Ontología es la rama de la Filosofía que trata de la naturaleza y organización de la realidad. En Inteligencia Artificial, y en particular en la comunidad de Ingeniería del Conocimiento, la palabra *ontología* se utiliza para denotar un objeto particular en vez de una disciplina genérica. Para los sistemas de Inteligencia Artificial lo que existe es aquello que puede ser representado [Gruber94]. La primera definición de ontología en el campo de la IA corresponde a [Neches *et al.* 91]:

> *" Una ontología define los términos y relaciones básicos que comprenden el vocabulario de un área de interés, además de las reglas de combinación de términos y relaciones para extender dicho vocabulario"*

Según esta definición, una ontología no sólo incluye los términos explícitamente definidos en ella sino también, de forma indirecta, los términos que se pueden inferir a partir de ellos usando ciertas reglas. Posteriormente se han propuesto numerosas definiciones en la literatura algunas de las cuales incluimos en este apartado. La definición propuesta por Gruber en 1993 es una de las más famosas y más referenciadas en la literatura:

> *"Una ontología es una especificación explícita de una conceptualización"* [Gruber93]

Conceptualización se entiende como una estructura semántica intensional (vs. extensional) que codifica las reglas implícitas que restringen la estructura de un fragmento de la realidad. Tomando como punto de partida la definición de Gruber, se han propuesto distintas modificaciones. Con el espíritu de reconciliar todas ellas, el trabajo de [Guarino&Giarreta95] recoge algunas definiciones y proporciona interpretaciones sintácticas y semánticas de ellas.

En la práctica el término ontología se utiliza de forma ambigua con dos significados, tanto para referirnos a componentes de nivel simbólico (sintáctico) o a su componente conceptual semántica equivalente. En [Guarino&Giarreta95] se propone usar el término *conceptualización* para denotar una estructura semántica y *teoría ontológica* para denotar una teoría lógica que expresa conocimiento ontológico. La intuición es que las teorías ontológicas son componentes concretos que pueden ser físicamente compartidas. Por otra parte, las conceptuali-

zaciones son el equivalente semántico de las teorías ontológicas. De este modo, la misma teoría ontológica puede satisfacer distintas conceptualizaciones, y la misma conceptualización puede ser la base de distintas teorías ontológicas. En este sentido, el término ontología se usa de forma ambigua a veces como sinónimo de teoría ontológica y otras como sinónimo de conceptualización.

Esta postura es compatible con la definición clásica de [Gruber93] si se considera el término ontología como sinónimo de teoría ontológica, ya que la aparición de la palabra "explícita" en la definición denota algo concreto y físico al nivel simbólico.

Además de las anteriores, existen otras definiciones en la literatura. Por ejemplo, la siguiente definición observa que lo que constituye una ontología es la interpretación semántica de los términos de un dominio y no los términos en sí mismos.

> *"Una ontología para el conocimiento relativo a una tarea o dominio particulares describe una taxonomía de conceptos para esa tarea o dominio que define la interpretación semántica del conocimiento"* [Alberts93]

El problema de la definición anterior es que no refleja que una ontología es mucho más que una taxonomía de conceptos, incluyendo entre otras cosas restricciones y relaciones entre los conceptos.

Desde el punto de vista de N. Guarino [Guarino97] la siguiente formulación clarifica la de [Gruber93] al hacer hincapié en el hecho de que las ontologías pertenecen al nivel de conocimiento, aunque podrían depender de puntos de vista particulares, reflejando el problema de interacción.

> *"Una ontología es una especificación explícita a nivel de conocimiento, de una conceptualización, [...] que puede estar influido por la tarea y el dominio particulares"* [Van Heijst *et al.* 97]

En 1994, Tom Gruber propuso otra definición que distingue las ontologías y las conceptualizaciones como cosas distintas, de forma que una ontología no es una especificación de una conceptualización sino "*un acuerdo (posiblemente incompleto) sobre una conceptualización*".

Por último, y para terminar con este breve repaso, la siguiente definición propuesta en [Swartout *et al.* 97] nos sirve para introducir la discusión terminológica en cuanto a la distinción entre los términos ontología y base de conocimiento.

> *"Una ontología es un conjunto de términos jerárquicamente estructurados para describir un dominio, que pueden ser utilizados como esqueleto en el que se fundamenta una base de conocimiento"*

En la comunidad de Ingeniería del Conocimiento algunas veces se utiliza la palabra ontología para referirse a una base de conocimiento dentro de un sistema. Según la definición anterior y según [Guarino&Giaretta95] una ontología no es una base de conocimiento ya que aunque ambas tienen en común que contienen conocimiento existen muchas diferencias entre ellas. Recogemos aquí las más representativas [Gómez-Pérez98]:

- Características del lenguaje utilizado para codificar el conocimiento. Las ontologías deben escribirse en un lenguaje procesable de forma automática, semánticamente bien definido, independiente del dominio, portátil, declarativo y expresivo, que debería ser independiente del lenguaje de la aplicación que reutiliza y comparte sus definiciones. Estas propiedades no se garantizan en general cuando se codifica una base de conocimiento.
- Objetivos de la codificación del conocimiento. Las ontologías están diseñadas con el propósito de compartir y reutilizar conocimiento, no siendo así en las bases de conocimiento. Por tanto, sus definiciones se deberían conceptualizar con un nivel alto

de abstracción y generalidad, para que las definiciones incluidas en la ontología sean independientes de su uso final.

- Especificación de requisitos. Uno de los problemas principales que los ingenieros de conocimiento tienen al construir bases de conocimiento es la dificultad de obtener los requisitos de la aplicación. Debido a que los expertos normalmente no son capaces de describir cómo se comportan en el dominio de forma concreta y completa, es muy difícil para los ingenieros de conocimiento especificar el comportamiento del KBS. Por tanto, los KBSs normalmente se construyen incrementalmente mediante prototipos que evolucionan de forma que las deficiencias del producto final de un ciclo se utilizan como especificación del siguiente prototipo. Por el contrario, las ontologías se construyen para ser reutilizadas y compartir su conocimiento en cualquier momento y situación independientemente de la aplicación que las utilice. Por tanto, los ontólogos son capaces de especificar, al menos de forma parcial, qué vocabulario será cubierto por la ontología en un dominio determinado. Esto constituye la principal diferencia entre el desarrollo de ontologías y de bases de conocimiento.

Resumiendo diremos que una ontología es una base de conocimiento reutilizable, y una base de conocimiento es una representación explícita, expresada en un lenguaje formal, del conocimiento acerca de un dominio.

Otro modo de estudiar la diferencia entre ontologías y bases de conocimiento es en términos de la caracterización funcional de su uso. Las siguientes características han sido extraídas de un mensaje enviado por Adam Farquhar del KSL (*Knowledge Sharing Laboratory*) de la Universidad de Stanford a una lista de distribución por correo electrónico. El conocimiento ¿Expresa el conocimiento consensuado de una comunidad de personas o de agentes? ¿Se usa como referencia de términos definidos de forma precisa? ¿El lenguaje es suficientemente expresivo para que la gente pueda expresar lo que quiere? ¿Puede ser reutilizado en distintas situaciones para distintos episodios de resolución de problemas? ¿Es estable? ¿Se puede usar para resolver distintos tipos de problemas? ¿Se puede usar como punto de partida para construir distintos tipos de aplicaciones incluyendo una nueva base de conocimiento, un esquema de base de datos o un programa orientado a objetos? Cuanto más positivas sean las respuestas a estas preguntas, más ontológico es el conocimiento representado.

El conocimiento de la mayoría de los KBSs se ubica en algún lugar entre los dos extremos. Los KBSs típicamente contienen parte de conocimiento ontológico y reutilizable pero normalmente influido por la tarea para la que se diseñó, las restricciones del lenguaje de representación, y los procedimientos de inferencia utilizados.

3.1.1 Conceptualización vs formalización

De lo anterior, podemos distinguir entre dos aproximaciones, la primera entiende una ontología como un marco conceptual al nivel de conocimiento y la segunda entiende una ontología como un componente concreto al nivel simbólico que será utilizado con algún propósito. Retomamos la terminología propuesta por [Guarino&Giarreta95] donde las teorías ontológicas son componentes concretos formalizados que pueden ser físicamente compartidos, y las conceptualizaciones son el equivalente semántico de las teorías ontológicas. En este sentido, el término ontología se usa de forma ambigua como sinónimo de teoría ontológica o como sinónimo de conceptualización.

En la línea de [Gruber93] y, como se defiende también en [Guarino&Giarreta95] y en [Chandrasekaran *et al.* 98], nosotros usaremos el término conceptualización en el sentido semántico y el término ontología como una teoría lógica que da una especificación explícita

(parcial) de una conceptualización. Para clarificar, en concreto entendemos las ontologías como un vocabulario de representación especializado en algún dominio o área de interés. En realidad, no es tanto el vocabulario concreto lo que caracteriza a una ontología sino las conceptualizaciones capturadas por los términos del vocabulario. Por ejemplo, la ontología en sí no cambiaría aunque traduzcamos los términos del vocabulario de inglés a español (porque no cambia su conceptualización).

En cualquier caso, después de la conceptualización de los principales componentes de una ontología, ésta debe implementarse en algún lenguaje que puede variar desde muy informal, semi-formal o lenguajes muy formales [Uschold96].

Después de la fase de conceptualización de un área de conocimiento, que requiere un análisis efectivo del área, se definen los términos asociados a las conceptualizaciones y se elige una sintaxis para codificarlos. El resultado de todo lo anterior se codifica en forma de una ontología del área de conocimiento elegida que puede ser compartida por cualquiera que tenga necesidades similares de representación de conocimiento en ese dominio evitando la necesidad de replicar el esfuerzo de análisis realizado.

3.1.1.1 Lenguajes de formalización de ontologías

Para *formalizar* el conocimiento que está presente en las ontologías se utilizan cinco tipos de componentes: clases, relaciones, funciones, axiomas e instancias [Gruber93]. Aunque las clases de una ontología se organizan generalmente de forma taxonómica, no se debe considerar esta taxonomía de clases como una ontología en sí misma [Studer *et al.* 98].

El término concepto o clase se utiliza en un sentido amplio para agrupar conjuntos de individuos, que pueden describir a su vez tareas, funciones, acciones, estrategias, procesos, etc. Las relaciones representan un tipo de interacción entre los conceptos del dominio. Las funciones son un caso especial de relación n-aria en el que el elemento n de la relación está unívocamente determinado por los n-1 anteriores elementos. Los axiomas se usan para modelar sentencias que son siempre ciertas en este dominio y las instancias se usan para representar individuos concretos.

Una vez que los componentes de la ontología hayan sido identificados, la ontología puede ser implementada en algún lenguaje. En las aplicaciones que usan ontologías podemos encontrar una gran variedad de lenguajes, por ejemplo, EXPRESS [Spibey91], CML el lenguaje de modelado de CommonKADS [Schreiber *et al.* 94], Ontolingua [Gruber93], KIF (*Knowledge Interchange Format*) [Genesereth&Fikes92], CycL[Lenat&Guha90], FLogic [Kifer *et al.* 95], o el propio LOOM [MacGregor&Bates87] [MacGregor88]. Cada uno de estos lenguajes se basa en alguno de los paradigmas de representación de conocimiento, como las DLs, los sistemas de marcos o la lógica de predicados.

Dentro de la comunidad de Ingeniería Ontológica, muchos trabajos consideran a Ontolingua como el estándar de lenguaje de especificación de ontologías. Ontolingua proporciona un lenguaje a nivel de conocimiento, independiente de la implementación, y que permite especificar conceptos, relaciones y axiomas. Ontolingua es una extensión de KIF y *Frame Ontology* [Gruber93], una ontología que captura las primitivas de representación utilizadas en los lenguajes de representación basados en marcos. Ontolingua no tiene un motor de inferencias asociado y la única manera de utilizar (por una máquina) una ontología escrita en Ontolingua es traducirla a algún lenguaje implementado. Como la traducción de conjuntos arbitrarios de axiomas en lógica a otros lenguajes no es viable, Ontolingua está sesgado hacia un estilo de representación orientado a objetos [Gruber93].

Debido al intercambio de ontologías a través de la Web, recientemente se han desarrollado nuevos lenguajes de especificación de ontologías basados en los estándar Web como

XML o RDF. Existen varios trabajos [Van Harmelen&Fensel99] [Gil&Ratnakar02] que presentan un estudio sobre ontologías y lenguajes basados en Web para representarlas. En [Corcho&Gómez-Perez00 a, b, c] se establece un marco común que permite comparar la expresividad de los lenguajes tradicionales de formalización de ontologías como LOOM, con los nuevos lenguajes más orientados al intercambio de ontologías a través de la Web. Por ejemplo, SHOE [Luke&Heflin00], XOL [Karp *et al.* 99], RDF [Lassila&Swick99] y OIL [Horrocks *et al.* 00]. Además de la expresividad compara también otros aspectos como la capacidad de inferencia y la facilidad de comprensión del código representado en dichos lenguajes. Aunque sólo concluyen que la elección del lenguaje más adecuado dependerá en cada caso de las características concretas la aplicación a desarrollar, los resultados obtenidos durante su estudio se pueden usar como guía para tomar esta decisión. En las tablas comparativas proporcionadas se reflejan las buenas propiedades de LOOM en los aspectos comparados. En el Apartado 6 describimos las DLs, y justificamos nuestra elección basándonos en la expresividad y los mecanismos de razonamiento que han hecho de las DLs un formalismo muy popular para formalizar ontologías.

3.1.2 Construcción de ontologías

Igual que el desarrollo de KBSs el desarrollo de ontologías también se encuentra, en cierto modo, con el cuello de botella de adquisición de conocimiento. Según la opinión de Van Heijst y sus colegas en [Van Heijst *et al.* 97] existen dos impedimentos que dificultan el desarrollo de nuevas ontologías. En primer lugar el problema de la interacción ya comentado, según el cual no se puede representar conocimiento si no se conoce cómo se va a razonar con él. En segundo lugar el problema de la gran cantidad de conocimiento en el mundo hace que la construcción de una ontología en una parcela concreta resulte una tarea desalentadora ya que el proceso de identificación de un conjunto de términos básicos reutilizables no resulta sencillo en absoluto.

Realizar un análisis ontológico permite clarificar la estructura del conocimiento y extraer las conceptualizaciones subyacentes que nos permitan disponer de un vocabulario para representar conocimiento de un cierto dominio. Por tanto, el primer paso en la representación de conocimiento sobre un dominio es llevar a cabo un análisis ontológico efectivo.

El proceso de construcción de ontologías es un *arte* más que una actividad de ingeniería [Gómez-Pérez98]. Cada equipo de desarrollo usa sus propios criterios de diseño y fases en el proceso de desarrollo de ontologías, la ausencia de metodologías estandarizadas dificulta el desarrollo de ontologías compartidas y consensuadas en y entre los equipos, la extensión de una ontología por otros y su reutilización en aplicaciones. Según [Gómez-Pérez98] la causa de estos problemas es la ausencia de un modelo conceptual explícito y totalmente documentado sobre el que formalizar la ontología.

Existen algunas aproximaciones metodológicas para construir ontologías. Por ejemplo, la metodología de Uschold [Uschold96] [Uschold&Grunninger96], la de Grüninger y Fox [Grüninger&Fox95] y METHONTOLOGY [Fernández *et al.* 99]. Ver también [Gruber95]. Todas ellas comienzan con una fase de identificación de los propósitos de la ontología y de la necesidad de adquirir conocimiento sobre el dominio. Después de la fase de adquisición de conocimiento, las metodologías reflejan el hecho de que los desarrolladores tienden a pasar directamente a la fase de implementación. Por ejemplo, según la metodología de Uschold después de adquirir cierto conocimiento del dominio se procede a su codificación en un lenguaje formal. La alternativa planteada en METHONTOLOGY propone expresar las ideas o conceptos usando una representación intermedia de forma que la ontología se puede formalizar posteriormente utilizando traductores a distintos lenguajes. Como hemos visto existen

varios sistemas de representación que se utilizan para formalizar ontologías, generalmente basados en marcos, en lógica o en ambos. La representación intermedia está a medio camino entre la visión del dominio por parte de la gente, en particular los expertos del dominio, y los lenguajes en los que se formalizan las ontologías. Dicha representación intermedia es una aproximación adecuada para adquirir y evaluar el conocimiento tanto para los ingenieros del conocimiento como para los expertos del dominio que no tienen por qué dominar el formalismo de representación elegido.

Uno de los objetivos del diseño de las ontologías es el de proporcionar un medio efectivo para compartir conocimiento. Después de realizar un análisis ontológico y obtener conceptualizaciones adecuadas para una cierto área de conocimiento, se eligen ciertos términos y una sintaxis para representarlas. Una vez formalizada, una ontología puede compartirse con cualquiera que se encuentre con necesidades similares de conocimiento en ese dominio, evitando el esfuerzo de repetir el proceso de análisis ontológico.

3.1.3 Clasificación de ontologías

En este apartado queremos dar una idea general de los tipos de ontologías que se manejan normalmente en la comunidad ontológica [Van Heijst *et al.* 97] [Mizoguchi *et al.* 95] [Chandrasekaran *et al.* 99]. La clasificación siguiente no es una partición exhaustiva ya que una ontología puede enmarcarse en varios de los siguientes tipos:

- Las *ontologías de representación de conocimiento* [Van Heijst *et al.* 97] capturan las primitivas de representación utilizadas para formalizar conocimiento en algún paradigma de representación. Explican las conceptualizaciones que subyacen en algún formalismo de representación de conocimiento y proporcionan un marco de representación neutral respecto a las entidades concretas del mundo o del dominio que se representarán con este tipo de ontologías. El ejemplo más representativo es la ya citada *Frame Ontology* [Gruber93] que captura las primitivas de representación utilizadas en los lenguajes de representación basados en marcos. Esta ontología permite especificar otros tipos de ontologías usando las convenciones de los sistemas de marcos.

- Las *ontologías de conocimiento general o de sentido común* capturan vocabulario relativo a cosas generales como eventos, tiempo, causalidad, espacio, comportamiento, funcionalidad, etc. En esta categoría el ejemplo más significativo es la ontología CYC [Lenat&Guha90] que incluye conocimiento humano básico de sentido común.

- Las *ontologías de nivel superior* (*Top-Level*) proporcionan elementos y términos con un alto nivel de abstracción, es decir, términos generales bajo los cuales se suelen colocar los términos más específicos de otras ontologías. Existen varios ejemplos de ontologías de nivel superior: *Penman Upper Level* [Bateman90], CYC [Lenat&Guha90], y Mikrokosmos [Mahesh96] son las más representativas. Relacionadas, se encuentran también las *meta-ontologías* u *ontologías genéricas* [Van Heijst *et al.* 97] (también conocidas como *Core Ontologies* o *Upper Ontologies* [Chandrasekaran *et al.* 99]) que son reutilizables a través de varios dominios. Uno de los ejemplos más típicos es la *Mereology Ontology* [Borst97] que define la relación `parte-de` y sus propiedades, lo que permite expresar que ciertos dispositivos están compuestos de partes que a su vez pueden descomponerse en subpartes. También en [Guarino95] y [Lenat&Guha90] se han propuesto ontologías genéricas alternativas, con un acuerdo en términos como "objetos", "propiedades", "valores", "relaciones", "procesos", "estados", "partes", "efectos", "clases", "instancias", "subclases", "componentes", etc. Este repertorio de términos no dice nada sobre qué clases o qué entidades u objetos existen realmente.

En ocasiones los términos de una ontología de un dominio se definen como especializaciones de los términos que pertenecen a este tipo de ontologías genéricas.

- Las *ontologías del dominio* [Van Heijst *et al.* 97][Mizoguchi *et al.* 95] proporcionan vocabulario sobre los conceptos de un dominio y las relaciones entre ellos, las actividades que se desarrollan y los principios elementales que rigen el comportamiento en ese dominio. Expresan conceptualizaciones que son específicas para un dominio y que se pueden reutilizar en distintas aplicaciones dentro de ese dominio. Las metodologías de Ingeniería del Conocimiento hacen una distinción explícita entre las ontologías del dominio y el conocimiento del dominio. El conocimiento del dominio describe situaciones reales en un cierto dominio, mientras que una ontología del dominio restringe la estructura y los contenidos del conocimiento del dominio.
- Las *ontologías lingüísticas* como Wordnet, EuroWordnet o Generalized Upper Model (GUM). Wordnet [Miller90] es una base de datos léxica para el idioma inglés cuya información se organiza en unidades llamadas *synsets* o conjuntos de términos sinónimos que son intercambiables en un contexto particular. Cada uno de estos conjuntos representa un significado distinto y todos los términos dentro de él comparten dicho significado. EuroWordnet [Vossen99] es una colección de bases de conocimiento léxicas o *wordnets*, cada una de las cuales representa los conceptos léxicos de un idioma y que se conectan a través de un índice interlingüa que define una relación de equivalencia entre los conjuntos de sinónimos de las distintas bases de conocimiento. Por último, GUM[1] es una ontología lingüística independiente del dominio y de las tareas que, para facilitar su portabilidad a varios idiomas, sólo incluye los conceptos lingüísticos principales y cómo se organizan en los distintos idiomas, y omite aquellos detalles que son distintos en idiomas distintos.
- Las *ontologías de tareas* [Mizoguchi *et al.* 95] proporcionan un vocabulario sistemático de los términos utilizados para resolver problemas asociados con tareas compartidas por distintos dominios. Relacionadas con éstas se encuentran las llamadas *ontologías de tareas-dominios* que son ontologías de tareas que sólo se pueden reutilizar en un cierto dominio y no en otros, debido a que la terminología asociada a la tarea está particularizada a un dominio concreto.
- Las *ontologías de métodos de resolución de problemas* proporcionan definiciones de los términos, conceptos y relaciones, que se utilizan para especificar un proceso de razonamiento llevado a cabo para lograr una tarea [Chandrasekaran *et al.* 99]. Según la definición de [Gennari *et al.* 98] proporcionan la estructura y el formato de las necesidades de conocimiento de los PSMs. Son independientes del dominio y sus elementos se describen en términos neutrales respecto a una aplicación a un dominio concreto. Por ejemplo, una ontología asociada a un método de diagnóstico no se referirá a "enfermedades", lo que la haría dependiente del dominio médico, sino a términos genéricos como "defecto" o "fallo".
- Las *ontologías de aplicación* [Van Heijst *et al.* 97] contienen el conocimiento y las definiciones necesarias para una cierta aplicación. En este tipo de ontologías típicamente se mezclan conceptos de ontologías de dominio y ontologías genéricas. Es más, pueden contener extensiones específicas de los métodos y tareas de la aplicación. Realmente, las ontologías de aplicación no son reutilizables en sí mismas. Se obtie-

[1] http://www.darmstadt.gmd.de/publish/komet/gen-um/newUM.html

nen a partir de la selección de ontologías de una biblioteca que se refinan para la aplicación concreta. Podemos encontrarlas, por ejemplo, en PROTÉGÉ-II.

Para construir una nueva ontología se pueden combinar ontologías de los distintos grupos. Por ejemplo, podemos comenzar decidiendo el paradigma utilizado para formalizar el conocimiento haciendo que la nueva ontología se ajuste a una ontología de representación de conocimiento. En el siguiente paso se decide si se incorporan ontologías de sentido común, en cuyo caso se construyen o reutilizan de alguna biblioteca y se añaden a la nueva ontología. En este punto comienza el proceso de modelado de los componentes del sistema. El conocimiento del dominio y el conocimiento de resolución de problemas puede modelarse simultáneamente. Durante el modelado de conocimiento del dominio primero se construirán (o reutilizarán) las ontologías genéricas, luego las específicas del dominio y finalmente las específicas de la aplicación. Durante el modelado de conocimiento de resolución de problemas, primero se modelan (o reutilizan) las ontologías de tareas y de métodos, después las ontologías de tareas-dominios. Las ontologías de métodos y de tareas permiten representar explícitamente el conocimiento necesario para evitar el problema de interacción entre las ontologías del dominio y los PSMs [Benjamins *et al.* 96] [Gómez-Pérez&Benjamins99].

3.2 Reutilización de Ontologías

Hemos visto que una idea subyacente común a las principales metodologías de soporte al diseño de KBSs es la separación entre distintas componentes de conocimiento para promover su reutilización. Se propone un tipo de construcción de KBSs donde resulta esencial que seamos capaces de compartir y reutilizar componentes construidos por otros. A pesar de las ventajas de compartir ese conocimiento y de la disponibilidad de esos componentes en bibliotecas [Neches *et al.* 91] [Patil *et al.* 92] [Farquhar *et al.* 95], importar y usar esos componentes no es una tarea fácil. La reutilización de ontologías no es un proceso automático y requiere un esfuerzo por parte del ingeniero del conocimiento. Además, no existen muchos artículos en la literatura que ejemplifiquen esta reutilización o que describan los detalles de cómo se aplican o qué aspectos afectan a la reutilización de ontologías existentes (aunque este aspecto puede verse por ejemplo en [Uschold *et al.* 98]).

El proceso de reutilización de una ontología requiere convertir la especificación a nivel de conocimiento que proporciona dicha ontología a una implementación concreta. Según [Uschold *et al.* 98] este proceso consume mucho tiempo y requiere considerar de forma cuidadosa el contexto, el uso deseado y las capacidades de representación tanto del lenguaje fuente de la ontología como del lenguaje destino de implementación, y la tarea específica de la aplicación concreta. También se argumenta que no todos los lenguajes de implementación proporcionan las mismas capacidades de razonamiento (ver también [Gruber93]) y que para decidir cómo restringir las capacidades de inferencia de la ontología ya implementada se debería usar como guía tanto la tarea específica como la funcionalidad de la aplicación.

En esta línea, y como veremos posteriormente, nuestra propuesta es usar DLs para representar –formalizar– el modelo de un dominio que se utilizará para resolver tareas asociadas al CBR, en concreto en sistemas KI-CBR. Diversos trabajos [Koelher94] [Kamp96] [Napoli *et al.* 96] [Napoli&Lieber96] [Napoli *et al.* 97] [Salotti&Ventos98] [Lieber&Napoli98], y en particular nuestra propia experiencia [Gómez *et al.* 98] [Gómez *et al.* 99] [Díaz&González00a] [Díaz&González01b] [Díaz&González01d], han probado que las DLs son una tecnología adecuada para la representación del conocimiento necesario para CBR. En ese sentido la reutilización de ontologías que utilizaremos en esta tesis se refiere a ontologías implementa-

das mediante DLs y utilizamos el término ontología como una especificación explícita de una conceptualización.

En [Gómez-Pérez98] se describe el proceso que un ingeniero de conocimiento lleva a cabo para desarrollar un KBS en el que se reutilizan ontologías. Una vez que el ingeniero tiene una idea más o menos clara del conocimiento que quiere modelar, la primera actividad supone seleccionar aquellas ontologías que son potencialmente útiles para su sistema. En general, no todas las definiciones que se incluyen en una ontología serán útiles para la aplicación concreta, por lo que el ingeniero de conocimiento debe seleccionar aquellos términos que resulten adecuados y proceder al proceso de traducción al lenguaje destino en el que se formalizará la base de conocimiento de la aplicación. En este punto el ingeniero identifica el subconjunto de la ontología completa que capture las ideas esenciales necesarias. De manera análoga, si el ingeniero de conocimiento ha identificado términos que no están incluidos en la ontología, deberá conceptualizarlos y formalizarlos. Finalmente, las dos partes se ensamblan para formar la base de conocimiento de la aplicación. La ventaja clara de este esquema es que el ingeniero sólo tendrá que incluir parte del conocimiento, en concreto sólo las definiciones que la ontología no proporcione.

En el proceso anterior, se pone de manifiesto el hecho de que es frecuente que no encontremos una ontología reutilizable que satisfaga exactamente los requisitos de la aplicación. Es por eso que se requiere el uso de varias ontologías que hay que integrar o el refinamiento de alguna de las ontologías reutilizadas, añadiendo y/o eliminando términos.

El término *integración* tiene diferentes acepciones dentro de la comunidad de Ingeniería Ontológica. Por ejemplo, en [Sowa00] se define la integración como el proceso de encontrar elementos en común entre dos ontologías diferentes A y B, y la derivación de una ontología C que facilite la interoperatividad entre sistemas que se basen en las ontologías A y B. La nueva ontología C puede reemplazar a las anteriores o utilizarse como intermediaria entre un sistema que se base en la ontología A y otro que se base en la ontología B.

En [Pinto *et al.* 99] se identifican tres acepciones y proponen tres términos distintos para ellas (integración, mezcla y uso, respectivamente):

1. Integración de ontologías para construir una nueva ontología reutilizando otras ontologías disponibles, de forma directa o a través de extensiones, especializaciones o adaptaciones. Es decir, en vez de construir una ontología desde cero hacerlo por reutilización de otras existentes.
2. Integración de ontologías mezclando diferentes ontologías sobre el mismo tema –o dominio– en una única que las unifique. Esto es común cuando existen varias ontologías sobre un mismo dominio, por ejemplo, terminología sobre medicina, que difieren en la terminología, significado, aspectos de formalización, etc. En este caso todas se integran o mezclan en una única que unifique sus definiciones.
3. Integración de ontologías en aplicaciones que las utilizan. En este sentido la integración es relativa a la ontología y a la aplicación que la usa, y no a integración entre ontologías. Este aspecto se trata en el siguiente apartado.

En la literatura se describen distintas alternativas y herramientas para la integración de ontologías, cada una con su acepción particular del término. Por ejemplo, citamos OBSERVER[2] [Mena *et al.* 00], Chimaera[3] [McGuiness *et al.* 00 a y b], FCA-MERGE [Stumme&Maedche01], ONIONS [Gangemi *et al.* 96] [Grosso *et al.* 98]. La integración de ontolo-

[2] http://siul02.si.ehu.es/OBSERVER/

[3] http://www.ksl.Stanford.EDU/software/chimaera/

gías es una línea de investigación activa no abordada en esta tesis. En nuestro trabajo suponemos un proceso de integración manual guiado por el usuario.

Chimaera es un sistema de soporte para la creación y el mantenimiento de ontologías distribuidas en la Web, así como a la integración (mezcla) entre distintas ontologías en distintos formatos de entrada. Chimaera acepta hasta 15 formatos, como ANSI KIF, Ontolingua, PROTÉGÉ, CLASSIC, iXOL y pronto aceptará otros estándares como RDF y DAML. El entorno da soporte a los usuarios en las tareas de abrir bases de conocimiento en distintos formatos, reorganizar taxonomías, resolver conflictos en nombres parecidos, visualización de ontologías y edición de términos. Por la relación con nuestro trabajo Chimaera se describe en el Apartado 3.3.2.

3.2.1 Uso de ontologías

Aunque las ventajas son notables, el número de aplicaciones reales que reutilizan ontologías para modelar el conocimiento de la aplicación es pequeño. Esto se debe principalmente a que la mayoría de las veces las ontologías se construyen para una aplicación concreta sin tener demasiada consideración con aspectos relativos a la compartición y reutilización de conocimiento. Además existen ciertos problemas que hacen difícil la reutilización de ontologías existentes en aplicaciones nuevas y que en parte, son la causa del reducido número de aplicaciones hasta el momento [Arpirez *et al.* 98][Gómez-Pérez&Benjamins99]: las ontologías están dispersas físicamente en servidores distintos, la formalización es distinta dependiendo del servidor en el que se almacene, las ontologías dentro del mismo servidor están a menudo descritas con distintos niveles de detalle, no existe un formato común para mostrar la información de las ontologías de forma que los usuarios puedan seleccionar cuál es la más adecuada para sus propósitos.

De todas formas existen bastantes aplicaciones que utilizan ontologías. Por ejemplo, las ontologías lingüísticas se usan frecuentemente en aplicaciones que hacen algún tratamiento del lenguaje natural. Las ontologías GUM y Wordnet se han utilizado en aplicaciones de generación del lenguaje natural en varios idiomas, la ontología *Enterprise Ontology* [Uschold96] se ha utilizado para construir un entorno llamado Enterprise tool set[4]. (Onto)2Agent es un entorno para seleccionar ontologías que satisfacen un conjunto dado de restricciones y *Chemical OntoAgent* es una aplicación para enseñar química [Arpirez *et al.* 98].

En cualquier caso es de esperar que el desarrollo y aplicación de las ontologías siga extendiéndose debido al papel preponderante que juegan en el proyecto de la Web semántica (*Semantic Web*[5]), una extensión de la Web actual en la que la información tiene un significado preciso y bien definido para facilitar el trabajo y la cooperación de ordenadores y personas. En sus orígenes la Web surgió como un espacio distribuido de información que diera soporte a la navegación a través de documentos relacionados y enlazados sobre Internet. Durante el gran crecimiento llevado a cabo desde su creación, la estructura de las páginas Web, que es crucial para hacer que la información sea legible por los ordenadores, se ha sacrificado a favor de los aspectos de visualización y presentación de los contenidos. La propuesta del proyecto de la Web Semántica se basa en el uso de una terminología común definida a través de ontologías está permitiendo la obtención de servicios más potentes como búsquedas, agentes software inteligentes y sistemas de gestión de conocimiento. Remitimos a los lectores

[4] http://www.aiai.ed.ac.uk/project/enterprise

[5] http://www.semanticweb.org/

interesados a trabajos recientes sobre la gestión de conocimiento en la Web a través de ontologías [Fensel01] [Kim01].

Ligado al proyecto de la Web Semántica se están desarrollando numerosos estándares que permitan la coexistencia de múltiples ontologías a través de documentos que hacen referencia explícita a las ontologías que están utilizando. Por ejemplo, SHOE (*Simple HTML Ontology Extensions*) incluye un conjunto de extensiones a HTML para anotar las páginas Web con conocimiento basado en ontologías sobre sus contenidos. Como ejemplo de las iniciativas que se están llevando a cabo en el desarrollo de ontologías para anotar documentos Web citamos SWRC (*Semantic Web Research Community Ontology*)[6] una ontología que modela la comunidad de investigación en la Web Semántica, incluyendo sus investigadores, publicaciones, temas de interés, herramientas, etc., así como las relaciones entre ellos. Esta ontología (disponible en varios formatos) es la base para anotar documentos, permitiendo así su acceso semántico. Otros esfuerzos de estandarización se están llevando a cabo para el dominio de las telecomunicaciones[7], para la representación de procesos de negocio[8] y para el desarrollo de vocabularios XML estandarizados para recursos humanos[9] entre muchos otros.

Además los sistemas de DLs juegan un papel fundamental en la Web Semántica, como formalismo para representar el conocimiento de las ontologías y para razonar con el, especialmente tras la unión DAML+OIL[10] [Horrocks02] [Horrocks&Patel-Schneider01] en un lenguaje Web para representar ontologías basado en DLs.

3.3 Herramientas de apoyo al uso de ontologías

Hoy en día se puede obtener información de organizaciones que ofrecen ontologías a través de Internet. Por ejemplo, las ontologías del Ontology Server[11] [Farquhar *et al.* 95] [Farquhar *et al.* 97] o WordNet[12] [Miller90] en Princeton están disponibles de forma gratuita. Otras ontologías como CyC[13] son parcialmente gratuitas; UCO (*Upper Cyc Ontology*)[14] es la parte de CyC disponible públicamente e incluye unas 3000 definiciones. En el proyecto Mikrokosmos del NMSU (*New Mexico State University*) *Computing Research Laboratory* se han desarrollado lexicones para español, japonés e inglés. De todas formas, debemos decir que la mayoría de las ontologías han sido desarrolladas por empresas para su uso particular y no están disponibles, al menos gratuitamente.

También existen algunas herramientas que ayudan a construir ontologías. Entre las más importantes se encuentran: Ontology Server, ODE (*Ontology Design Environment*), WebODE y Ontosaurus[15]. Además, las propias metodologías de desarrollo de KBSs incluyen sus propias herramientas. Por ejemplo, KADS, CommonKADS y PROTÉGÉ proporcionan herramientas de soporte durante todo el proceso de desarrollo basado en componentes de KBSs.

---

[6] http://ontobroker.semanticweb.org/ontos/swrc.html

[7] http://www.dmtf.org

[8] http://www.bpmi.org

[9] http://www.hr-xml.org

[10] http://www.daml.org/2001/03/daml+oil-index

[11] http://www-ksl-svc.stanford.edu:5915/ o http://granvia.dia.fi.upm.es:5915/ que corresponde a su mirror europeo en la Universidad Politécnica de Madrid

[12] http://www.cogsci.princeton.edu/~wn/

[13] http://www.cyc.com

[14] http://www.cyc.com/cyc-2-1/

[15] http://www.isi.edu/isd/ontosaurus.html

El entorno de desarrollo de KBSs PROTÉGÉ-II [Puerta *et al.* 92] basado en seleccionar y modificar PSMs y ontologías reutilizables, proporciona un generador de herramientas de adquisición de conocimiento basado en ontologías. Después de definir una ontología, la herramienta genera una interfaz gráfica para adquirir el conocimiento descrito por la ontología. En [Fensel *et al.* 99] se utiliza esta herramienta para desarrollar una meta-ontología para UPML (*The Unified Problem-solving Method Development Language*) [Fensel *et al.* 98a] y un editor basado en PROTÉGÉ para escribir las especificaciones de los métodos.

Recientemente se han utilizado ontologías en el entorno Web para distintos dominios. Por ejemplo, Ontobroker[16] [Fensel *et al.* 98b] es una herramienta para gestión de conocimiento que permite llevar a cabo la inspección avanzada de distintas fuentes de información Web. Proporciona visualización hiperbólica, una interfaz de consultas, un motor de inferencias para derivar las respuestas y un buscador para adquirir conocimiento en la Web. Otras herramientas disponibles pueden ayudar a seleccionar ontologías, por ejemplo la página conocida como TOP (del inglés, *The Ontology Page*)[17] o (Onto)2Agent[18].

El *Ontology Server* es el entorno de desarrollo más utilizado para construir ontologías en el lenguaje Ontolingua. Proporciona acceso a una biblioteca de ontologías que es una de las fuentes de conocimiento utilizadas por nuestra aplicación, por lo que se describe con mayor detalle en el apartado siguiente.

ODE (*Ontology Design Environment*) es un entorno de desarrollo de ontologías desarrollado en la Facultad de Informática de la Universidad Politécnica de Madrid. WebODE [Arpirez *et al.* 01] es la evolución de ODE y la versión cliente-servidor de la aplicación. Ambos están relacionados con la metodología METHONTOLOGY [Fernández *et al.* 99] ya que la principal innovación planteada en este entorno es que permite el desarrollo de ontologías al nivel de conocimiento y utiliza un formato de representación intermedia que es independiente del lenguaje final en el que implementará la ontología. Una ventaja de trabajar a este nivel es que los expertos del dominio que no son expertos en los lenguajes de formalización, pueden usar este entorno para diseñar, especificar y validar ontologías. Después de la fase de conceptualización de una nueva ontología el entorno ofrece generadores automáticos de código a distintos lenguajes.

Ontosaurus es un entorno desarrollado por el ISI (*Information Sciences Institute*) en la Universidad de Southern California, que tiene dos partes: un servidor de ontologías que usa LOOM como sistema de representación de conocimiento y un sistema de visualización de ontologías basado en páginas HTML que se generan de forma dinámica. Este entorno, igual que el Ontology Server, utiliza formularios HTML para la edición de ontologías, y ofrece traductores desde LOOM a distintos lenguajes como Ontolingua, KIF, KRSS e incluso C++.

3.3.1 Ontology Server y Ontolingua

El Ontology Server [Farquhar *et al.* 95] [Farquhar *et al.* 97] es el editor más utilizado para desarrollar ontologías en el lenguaje Ontolingua y fue desarrollado en el contexto del ARPA *Knowledge Sharing Effort* por el KSL (*Knowledge Sharing Laboratory*) de la Universidad de Stanford.

La idea subyacente es dotar de un conjunto de herramientas y servicios que den soporte para que distintos grupos de trabajo que estén geográficamente distribuidos puedan construir

[16] http://www.aifb.uni-karlsruhe.de/www-broker

[17] http://www.medg.lcs.mit.edu/doyle/top

[18] http://delicias.dia.fi.upm.es/REFERENCE_ONTOLOGY/

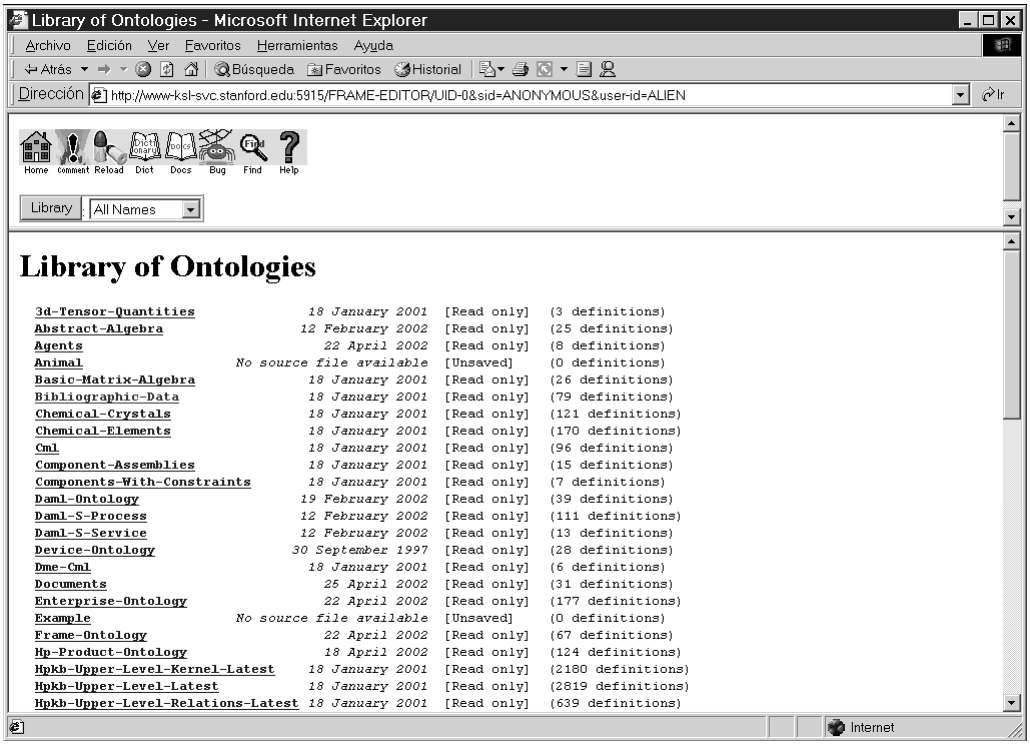


Figura 3-1. *Biblioteca de ontologías del Ontology Server*

y compartir ontologías. Proporciona un entorno de edición basado en el uso de páginas y formularios Web y visualización basada en HTML.

Además sirve como entorno de acceso a una biblioteca de ontologías desarrolladas por otros grupos (Figura 3-1). Fue esta biblioteca junto con su conjunto de traductores a lenguajes de representación tan conocidos como Prolog, IDLs de CORBA, CLIPS y el propio LOOM, la razón de elegir el Ontology Server como fuente principal de conocimiento ontológico para nuestro entorno de desarrollo de sistemas CBR (que se describe en el Capítulo 6).

Además, la biblioteca incluye diversas ontologías de distintos temas así como la posibilidad de diseñar nuevas ontologías y añadirlas a la biblioteca de forma local, o solicitar su inclusión en la biblioteca general que dependerá de la satisfacción de ciertos criterios de "calidad ontológica".

3.3.2 Chimaera

Chimaera[19][McGuiness *et al.* 00a y b] es un entorno Web de ayuda a la integración de ontologías construido sobre el Ontology Server. Entre sus funciones está la ayuda para reorganizar la taxonomía y resolver conflictos de nombres en una base de conocimiento. Además de ser una herramienta de ayuda para integrar ontologías el entorno también es útil para la visualización de ontologías. Chimaera incluye un entorno de edición de ontologías simple aunque ofrece la posibilidad de utilizar la funcionalidad más completa del Ontology Server.

Su principal característica es que facilita la integración de ontologías permitiendo a los usuarios que carguen ontologías previamente desarrolladas en una nueva base de conoci-

[19] http://www.ksl.Stanford.edu/software/chimaera/

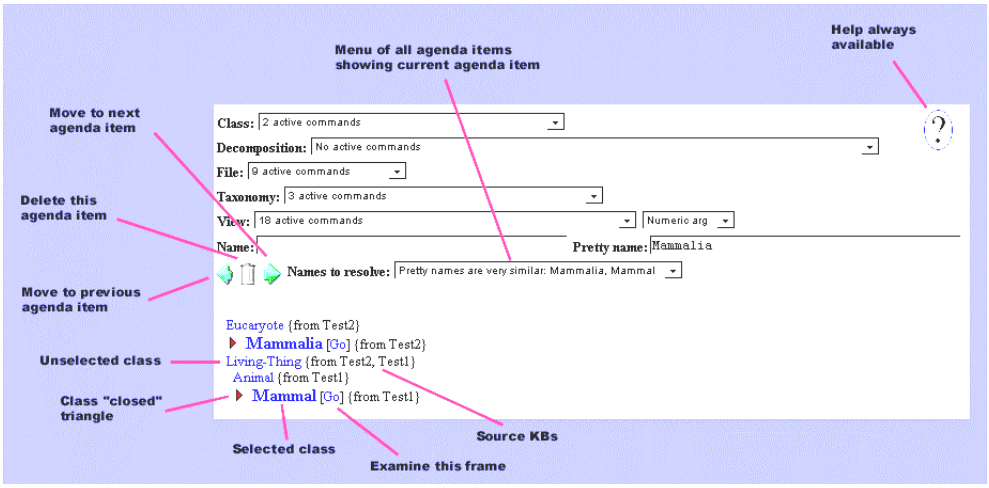


Figura 3-2. *El entorno Chimaera*

miento o en alguna previamente cargada. Para cargar una base de conocimiento en la actual, se pueden utilizar varias posibilidades y orígenes, por ejemplo desde la máquina cliente (usando el sistema de archivos del cliente), desde una URL, desde el sistema de archivos del servidor de Chimaera (sólo con permisos especiales) o desde el Ontology Server. En todos los casos, el contenido de la base de conocimiento debe ser código fuente correcto en alguno de los lenguajes de entrada de Chimaera.

La Figura 3-2 [McGuiness *et al.* 00b] muestra el resultado de cargar dos ontologías (Test1 y Test2) y elegir el modo de resolución de nombres. Chimaera sugiere candidatos potenciales a ser "mezclados" en un único término, utilizando sus propiedades. Genera una lista de resolución de nombres que se usa como guía durante la integración de ontologías. En el ejemplo, la lista de resolución de nombres incluye como sugerencia la mezcla de los conceptos "Mamal" y "Mammalia" debido a la similitud sintáctica de sus nombres. Además, el usuario puede ver el lugar en el que los dos términos aparecen en la jerarquía. Aunque sólo se visualiza un fragmento, el usuario puede visualizar más detalles expandiendo las subclases de cada uno de ellos (en el ejemplo los triángulos indican que los términos no tienen subclases). Además el usuario puede ver la definición de los términos y los resultados de similitud y diferencias de las comparaciones estructurales entre las definiciones (sólo para Ontolingua). El usuario decidirá si los términos se mezclan o no.

Después de la descripción de los aspectos generales relacionados con las ontologías el siguiente apartado describe el otro componente reutilizable que interviene en el proceso de desarrollo de KBSs y en nuestra aproximación: los métodos de resolución de problemas.

4. Métodos de resolución de problemas

Como hemos descrito en los apartados anteriores, entendemos una ontología como una especificación explícita de una conceptualización, es decir, como un vocabulario, o las conceptualizaciones capturadas por sus términos, que recoge conocimiento sobre algún dominio o área de interés. En este apartado, y en la línea de [Chandrasekaran *et al.* 98], proponemos ampliar la interpretación del término *conocimiento* para que incluya también conocimiento sobre la resolución de problemas. De esta forma, queremos introducir los conceptos básicos

y el trabajo relativo a ontologías de métodos de resolución de problemas. En general cuando hablamos de ontologías consideramos que definen conocimiento del dominio a un nivel genérico, mientras que los PSMs especifican conocimiento sobre mecanismos de razonamiento genéricos, que se pueden aplicar a distintos dominios y situaciones particulares. Las ontologías y los PSMs se consideran componentes reutilizables complementarios para construir KBSs a partir de componentes.

La noción de PSM está presente en las principales metodologías de Ingeniería del Conocimiento y durante más de diez años diversos grupos de investigadores en KBSs [Chandrasekaran86][Benjamins95][Benjamins&Fensel98] han considerado las tareas de definir, organizar, catalogar o clasificar PSMs en bibliotecas que pudieran ser reutilizadas por los desarrolladores de sistemas. Para ello estas bibliotecas se definen en torno a una ontología de PSMs, que como vimos en el Apartado 3.1.3, proporcionan las definiciones terminológicas que se utilizan para especificar los procesos de razonamiento llevados a cabo en los PSMs.

Son numerosos los beneficios derivados del uso de estas bibliotecas de métodos reutilizables, principalmente un menor tiempo de desarrollo y de mantenimiento ya que los métodos han sido previamente probados. Sin embargo, la construcción de una biblioteca de métodos que soporte reutilización de forma efectiva ha resultado ser un objetivo difícil.

Los PSMs definen el proceso de razonamiento de un sistema basado en conocimiento de forma independiente del dominio y de la aplicación. Describen el comportamiento de resolución de un problema al nivel de conocimiento en términos de las tareas a realizar y los objetivos a conseguir, las acciones necesarias para conseguir los objetivos y el conocimiento necesario para realizar las acciones.

Ya que los PSMs juegan un papel fundamental en la ingeniería y adquisición del conocimiento, en el siguiente apartado queremos dar una visión general de los aspectos principales relacionados con su representación y reutilización. El Apartado 4.2 define ciertos aspectos relativos a la reutilización de PSMs.

4.1 Estructura de los métodos de resolución de problemas

Antes de describir la estructura básica utilizada para representar los PSMs, es necesario introducir cierta terminología:

- Una tarea describe algún objetivo que se quiere cumplir. El diagnóstico es un ejemplo de tarea cuyo objetivo es encontrar todas las causas que expliquen un conjunto de síntomas. Las tareas se pueden descomponer recursivamente en subtareas.
- Un PSM describe cómo llevar a cabo una tarea, descomponiéndola en subtareas y/o pasos de inferencia, y especificando el conocimiento de control. Los pasos de inferencia son los operadores básicos sobre el conocimiento del dominio y el conocimiento de control define el orden entre estos operadores. Los PSMs típicamente dependen de la tarea que resuelven, es decir, se definen y se usan para resolver una tarea específica, y se organizan en bibliotecas donde cada tarea tiene asociada uno o varios métodos alternativos para resolverla. Aunque existan varios métodos capaces de resolver la misma tarea, la adecuación de uno u otro método a un contexto de aplicación concreto puede ser distinta.
- El modelo del dominio es el conocimiento real del que disponemos en un cierto momento. Puede crearse a partir de ontologías y particularizarse o no, para una apli-

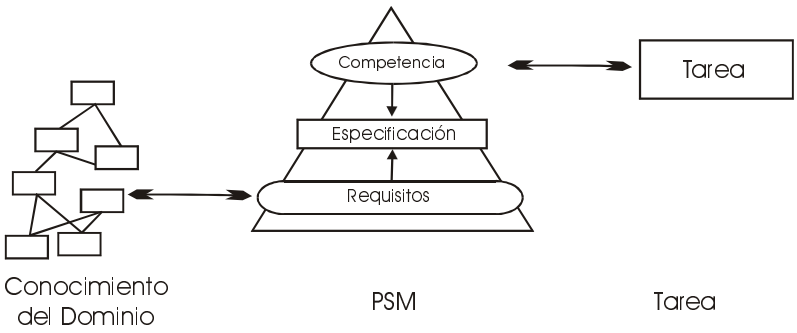


Figura 3-3. *Componentes de un PSM y su contexto de aplicación*

cación concreta e incluye el subconjunto de conocimiento de un dominio que es relevante para una cierta tarea.

4.1.1 Componentes de un método de resolución de problemas

Existe un acuerdo general sobre que un PSM está formado por tres partes relacionadas:[Benjamins *et al.* 96][Benjamins&Fensel98][Fensel&Motta98]

- La *competencia* o *especificación funcional* de un PSM es una descripción declarativa del comportamiento de entrada/salida del método y describe *qué* consigue el método, es decir, la tarea que es capaz de resolver.
- Los *requisitos* describen el conocimiento del dominio que un PSM necesita para conseguir su competencia, o dicho de otro modo, lo que el PSM espera a cambio de la competencia que ofrece. Por ejemplo, un método cuya competencia sea la tarea de obtener el diagnóstico que explique un conjunto de síntomas puede requerir la existencia de modelos causales o de fallo.
- La *especificación operacional* describe el *cómo*, es decir, el proceso de razonamiento que obtiene como resultado la competencia del método si dispone de los requisitos adecuados. Consiste en ciertos pasos de inferencia y el flujo de control entre ellos, es decir, su orden de ejecución. Los pasos de inferencia se describen por su relación entrada/salida y pueden llevarse a cabo por un paso de razonamiento primitivo atómico o por otro método, lo que significa que el PSM se descompone jerárquicamente.

4.2 Reutilización de PSMs

La reutilización de PSMs no siempre es algo sencillo y supone plantear entre otras las siguientes cuestiones: ¿De qué métodos dispongo? ¿Cómo se organizan los métodos disponibles en una biblioteca? ¿Cómo encontrar el método adecuado a la tarea que quiero resolver? ¿Podemos dar soporte al proceso de adaptar un método genérico a las circunstancias específicas de nuestra aplicación?

Los PSMs reutilizables se agrupan en bibliotecas y normalmente se organizan en torno a las tareas que resuelven. El siguiente apartado introduce algunas bibliotecas de métodos disponibles en la literatura. Sin embargo, el aspecto más importante para reutilizar los PSMs de una biblioteca durante la construcción de un KBS, se refiere a cómo tener el cuenta el *contexto* en el que queremos aplicar un cierto método. Como los PSMs están diseñados como com-

ponentes genéricos y reutilizables, es posible que no sean totalmente adecuados para un cierto contexto. Los PSMs resuelven tareas aplicando cierto conocimiento del dominio, por tanto, el contexto externo de un PSM estará formado por la tarea que lleva a cabo y por el conocimiento del dominio que aplica.

Relacionados con su contexto externo, existen varios factores o conflictos que dificultan la reutilización de un PSM. Uno de ellos es el propio problema de interacción ya comentado, según el cual una estrategia de razonamiento no se puede describir sin conocer en qué conocimiento del dominio será aplicado y recíprocamente que el conocimiento del dominio no se puede representar sin saber qué tipo de razonamiento lo utilizará. Este problema se solventa al explicitar las dependencias entre el método y el conocimiento del dominio en forma de *requisitos*.

Existen otros conflictos que dificultan la reutilización de métodos. Para resolverlos debemos estudiar cuáles son las causas que provocan que existan *huecos* entre un método y el dominio o un método y la tarea que se pretende que lleve a cabo. En ambos sentidos, hacia el dominio y hacia la tarea (ver Figura 3-3) el PSM puede estar definido utilizando una terminología distinta a la utilizada en el modelo de conocimiento o en la tarea. Por ejemplo, en un cierto PSM cuya tarea sea la obtención de un diagnóstico se requiere que los síntomas se ordenen según el valor de su atributo `grado`, pero en los síntomas del modelo de conocimiento no existe un atributo `grado` sino otro parecido llamado `nivel`. Para resolver el problema mencionado, se podría llevar a cabo un proceso de renombrado de términos que eliminara estas diferencias o un proceso de transformación que realizara una conexión o *mapping* entre los términos del modelo de conocimiento y los datos con los que trabaja el método. Estos datos definen los requisitos que debe satisfacer el conocimiento para que el método sea aplicable. En el ejemplo, el método requiere la existencia del atributo `grado`.

La distancia que puede existir entre el método y la tarea puede deberse a que la competencia ofrecida por el método no sea suficiente para resolver la tarea requerida, en cuyo caso habría que realizar simplificaciones sobre la tarea que se va a realizar. Desde el método hacia el conocimiento, si el modelo del dominio disponible no satisface los requisitos de conocimiento del método deberá completarse de manera adecuada, bien por integración, bien añadiendo el conocimiento adicional necesario.

La mayor parte del trabajo en reutilización de métodos se enmarca dentro de una tarea específica, es decir, los componentes reutilizables se especifican por tarea (planificación, diseño, valoración, diagnóstico). Sin embargo, existe otra línea de trabajo que amplía el ámbito de aplicación de los PSMs para reutilizarlos en distintas tareas. Sin embargo, el hecho de que los PSMs normalmente se definan para resolver una tarea concreta dificulta la reutilización de los métodos para resolver otras tareas ya que la descripción de los mismos utilizará terminología especifica de esa tarea [Beys *et al.* 96]. Por ejemplo, un método para la tarea diagnóstico usa ciertos términos específicos de esa tarea como `síntoma`, `causa` o `hipótesis`. Para reutilizar métodos a través de distintas tareas se deben identificar patrones comunes de comportamiento para definir PSMs independientes del contexto de aplicación, en el sentido de que no utilizan terminología que se refiere a una tarea específica [Beys *et al.* 96] [Fensel *et al.* 97]. Esta aproximación describe los procesos de razonamiento a un nivel más general y reutilizable, lo que los hace menos aplicables en el sentido de que existe más distancia entre el método y el contexto concreto en el que se quiere aplicar, y es más difícil hacer el *mapping* entre el método y el conocimiento del dominio que necesita. En [Fensel *et al.* 97] se plantea una caracterización de los métodos de forma independiente de la tarea definiéndolos como estrategias de búsqueda y elimina la distancia entre el método y la tarea concreta que se pre-

tende resolver usando una ontología de adaptación que permite reformular el método en términos específicos de la tarea.

Nuestra aproximación en esta tesis no define métodos reutilizables en varias tareas sino que los métodos que proponemos son específicos y resuelven alguna tarea CBR concreta y la reutilización de un método sólo tiene sentido en el contexto de la tarea para la que se definió. Además, los métodos se definen en términos CBR, por ejemplo, recuperar *casos*, y no en términos del dominio de aplicación.

4.2.1 Bibliotecas de PSMs

En los últimos años se han desarrollado numerosas bibliotecas de PSMs reutilizables, como las presentadas en [Marcus88] [Chandrasekaran *et al.* 92] [Puppe93] [Breuker&vandeVelde94] [Benjamins95] [Swartout *et al.* 97] [Valente *et al.* 98] y [Motta98].

La reutilización de bibliotecas de PSMs se ha ejemplificado en varios trabajos. En [Speel&Aben97] se ejemplifica el uso de la biblioteca de PSMs asociados a tareas de diagnóstico de [Benjamins95] en la construcción de un resolutor de problemas reales de la empresa Unilever. De especial importancia en este contexto es el proyecto IBROW [Benjamins *et al.* 99] cuyo objetivo es el desarrollo de agentes inteligentes que sean capaces de configurar componentes reutilizables de la Web para diseñar KBSs. El proyecto integra bases de datos distribuidas y tecnología Web con tecnologías de KBSs, principalmente ontologías y PSMs. La idea es que un agente software pueda seleccionar y combinar PSMs de distintas bibliotecas para ofrecer al ingeniero de conocimiento un mecanismo de soporte semiautomático para configurar un KBS. Para ello, es imprescindible disponer de un lenguaje de descripción de PSMs que proporcione descripciones de alto nivel, comprensibles por el ingeniero, que se basen en representaciones formales. Así, como parte de IBROW se ha desarrollado UPML (*The Unified Problem-solving Method Development Language*), un lenguaje para describir e implementar KBSs que se basan en reutilizar y configurar PSMs genéricos. El objetivo de UPML es la descripción e implementación de los PSMs para facilitar su reutilización semiautomática y dentro del proyecto IBROW3[20] se ha construido y se ofrece una biblioteca de métodos reutilizables.

El lenguaje UPML tomó como punto de partida el lenguaje CML desarrollado en el proyecto CommonKADs [Schreiber *et al.* 94], pero refinándolo convenientemente para ajustarse al estilo orientado a componentes de las arquitecturas software. En CML se considera un modelo conceptual en capas de los KBSs distinguiendo entre las capas del dominio, de inferencia y de tareas, según la metodología CommonKADs. En general, UPML es un lenguaje más formal que CML y que está mas orientado a la reutilización de PSMs.

Los cuatro componentes de una especificación UPML son: las tareas, que definen los problemas que resuelven los KBSs, los PSMs, que definen los procesos de razonamiento de forma independiente del dominio, los modelos del dominio, que describen el conocimiento del dominio del KBSs, y las ontologías que proporcionan la terminología utilizada en las tareas, los PSMs y las definiciones del dominio. Cada uno de estos componentes se describe de forma independiente para permitir su reutilización en diferentes contextos. Para integrar los distintos componentes se requiere el uso de dos tipos de *adaptadores*. Los *puentes* modelan explícitamente la relación entre dos partes de la arquitectura, por ejemplo, el dominio y la tarea, la tarea y el PSM, o el dominio y el PSM. El otro tipo de adaptadores (*refiners*) se usa

[20] http://www.swi.psy.uva.nl/projects/IBROW3/home.html

para especializar una clase de elementos de una especificación, en el caso de que se manejen componentes (principalmente tareas y métodos) genéricos.

La idea es que UPML no sea tanto un formalismo de estandarización sino una arquitectura estándar definida por una meta-ontología. Los seis bloques constructivos de UPML definen una arquitectura software y en [Fensel *et al.* 99] se describe la metaontología de UPML. La aproximación es la misma que se utiliza en Ontolingua [Gruber93] que define una meta-ontología para describir ontologías basadas en los sistemas de marcos.

En [Motta&Zdrahal96] se presentan cinco versiones de un método, llamado *Propose&Revise* aplicado a un dominio concreto para ilustrar la reutilización de PSMs en el nivel de conocimiento. En [Fensel *et al.* 97] se presenta una versión de *Propose&Revise* independiente de la tarea y luego lo aplica a la tarea de diseño paramétrico. El método *Propose&Revise* puede ser considerado como una clase de métodos más que como un método concreto, que permite distintos mecanismos de control y estrategias de revisión. La idea básica subyacente a este método es la de guiar los procesos de búsqueda utilizando mecanismos de vuelta atrás (*backtracking*) basados en conocimiento. Es decir, en vez de volver al último punto de elección, o usar dependencias, la resolución basada en *Propose&Revise* reacciona y soluciona las inconsistencias utilizando conocimiento que es específico de la aplicación concreta. Esto elimina la necesidad de los mecanismos de fuerza bruta o vuelta atrás ciega y mejora el rendimiento del resolutor.

Respecto a la organización de métodos en las bibliotecas, en la mayoría de las propuestas los métodos de una biblioteca se organizan por diseño, es decir, de manera estática por ejemplo en función de una cierta tipología, como en CommonKADS. En el sistema EXPECT [Swartout *et al.* 97] se plantea una alternativa a la organización estática de las bibliotecas de métodos según su tipo y proponen una biblioteca que se autoorganiza en torno a las capacidades de los métodos. La representación adecuada de la competencia de los métodos, utilizando LOOM, permite que el sistema pueda razonar con las descripciones usando subsunción y reformulación. Los métodos se organizan en un retículo de conceptos y, el clasificador semántico de términos de LOOM es capaz de encontrar la posición relativa de un método respecto a los otros métodos de la biblioteca razonando con la descripción de su competencia, es decir de los objetivos que consigue. Además, dado un objetivo o tarea a resolver, su aproximación semántica permite seleccionar un método adecuado, es decir, un método cuya competencia subsuma al objetivo requerido. Este tipo de representación también es adecuada para derivar o encontrar qué conocimiento utiliza un método en lugar de requerir que el diseñador deba especificarlo de forma estática.

4.2.2 Los *mappings* como mecanismo para integrar el conocimiento del dominio con los PSMs

Incluso si hemos seleccionado un método adecuado que resuelve la tarea que queremos, hemos visto que la reutilización de PSM genéricos conlleva ciertas dificultades que se deben principalmente al grado de independencia de los componentes reutilizables, por un lado el conocimiento declarativo del dominio y por otro los PSMs que razonan con ese conocimiento. Por tanto, es necesario disponer de algún mecanismo de integración tanto sintáctico como semántico que ayude a eliminar los posibles *huecos* existentes entre el conocimiento del dominio y el método (ver Figura 3-3).

Dada una tarea específica, un PSM y cierto conocimiento del dominio, las distintas metodologías de desarrollo de KBSs proponen soluciones distintas para integrar el método con el dominio. Por ejemplo, algunas soluciones están basadas en esquemas evolutivos y otras en

técnicas de adquisición de nuevo conocimiento o en técnicas de refinamiento del método. Nosotros estamos interesados en aquellas aproximaciones que suponen que las componentes a reutilizar han sido modeladas previamente y de forma independiente, y que se integran usando una capa intermedia de conversión.

Por ejemplo, el marco presentado en [Park *et al.* 98] dentro de la metodología PRÓTEGÉ, considera que las ligaduras entre el conocimiento del dominio y los componentes de los métodos consiste en la generación de conexiones entre los conceptos del dominio y conceptos análogos en el universo de discurso del método, es decir que pertenecen a la ontología correspondiente al método. Definen el concepto de relaciones de conexión declarativas que son especificaciones explícitas para las conexiones, tanto sintácticas como semánticas, entre las entidades en el conocimiento y las componentes del método. Estas relaciones se estructuran en una ontología que describe el rango de tipos de conexiones (*mappings*) declarativas que soporta PRÓTEGÉ para eliminar la distancia entre las clases de la ontología del dominio y los métodos. De esta forma las re laciones de conexión en PRÓTEGÉ son entidades concretas y formales que pertenecen a un conjunto de tipos preestablecidos.

En [Gennari *et al.* 98] la tarea de *mapping* se refiere a acercar las bases de conocimiento a los PSMs. Proponen un lenguaje de descripción de métodos cuyas anotaciones semánticas dan soporte a la tarea de modificar una cierta base de conocimiento para que satisfaga los requisitos de conocimiento del método.

En general, el término *mapping* entre dos componentes se refiere a cualquier mecanismo utilizado para realizar la conversión entre las estructuras que existen en uno de los componentes y las estructuras análogas esperadas por el otro componente. En nuestra aproximación estamos interesados en la conexión entre las estructuras que existen en el modelo de conocimiento del dominio y las estructuras que "espera" el PSM. El uso de estas relaciones de conexión entre la base de conocimiento y los métodos de resolución de problemas definidos de forma independiente del dominio, es uno de los pilares de nuestra aproximación.

En [Park *et al.* 98] se proponen tres tipos de relaciones de conexión:

- Los *mappings implícitos*, también conocidos como *mappings de adaptación de componentes*, son las conversiones que alguien realiza por especialización de alguno o ambos componentes para hacer que las definiciones de los objetos de uno satisfagan los requisitos del otro. Por ejemplo, la modificación de software genérico para que funcione sobre nuestras estructuras de datos concretas es una forma de *mapping* implícito. En general, consisten en cualquier modificación o particularización de un elemento que exista previamente. En nuestra consideración dicotómica de modelo de conocimiento y método de resolución, cualquiera de los dos componentes podría ser adaptado, el modelo de conocimiento para que satisfaga las necesidades del método o el método para que haga referencia a los términos del modelo del dominio.

  La principal ventaja de esta aproximación es su simplicidad conceptual, aunque tiene problemas principalmente respecto al mantenimiento, debido a que los componentes resultantes son específicos de la aplicación concreta. Cada vez que se reutiliza un cierto componente se produce una versión distinta de la implementación del método o del modelo del dominio, cada una de las cuales tiene que mantenerse de forma independiente. Otro problema puede deberse a que la especialización de componentes introduzca errores ya que las modificaciones normalmente se realizan *ad hoc* y no utilizan buenos principios metodológicos.

- Los *mappings procedimentales* consisten en código de traducción que convierte las instancias del modelo del dominio a los tipos requeridos por el método de resolución.

El código incluye conocimiento sobre qué instancias del dominio debe hacer corresponder con qué requisitos del método.

La principal ventaja de esta aproximación es su carácter procedimental que la hace directa, eficiente en ejecución y no modifica ninguno de los componentes originales. Entre las desventajas se encuentra que los módulos de traducción son específicos de la aplicación, de la tarea, del método y del modelo del dominio y resulta muy difícil su reutilización posterior. Además, el código puede no resultar comprensible para los expertos del dominio y produce problemas de mantenimiento igual que otro tipo de código.

- Los *mappings declarativos* constituyen un método descriptivo para definir las conversiones entre entidades de las componentes [Gennari *et al.* 94]. Consisten en dos módulos, un conjunto de relaciones de *mapping* y un intérprete para ellas. Las relaciones de *mapping* son especificaciones explícitas que definen las distintas conversiones que pueden llevarse a cabo para traducir objetos entre el modelo de conocimiento y el método. El intérprete analiza las declaraciones de las relaciones y realiza la conversión en tiempo de ejecución que proporciona las entradas al método.

  Una ventaja es que la implementación del intérprete es genérica y reutilizable para otras aplicaciones. Además, el carácter declarativo de las relaciones permite separar entre el *qué* de la transformación y el *cómo* que está oculto en el intérprete, lo que permite mayor claridad en las descripciones del diseñador.

En nuestro sistema, como detallaremos en el capítulo siguiente, la base formal de los *mappings* entre el modelo de conocimiento y los PSMs es el mecanismo de clasificación semántica del sistema de DLs y el uso de una terminología específica de CBR que se usa de forma consensuada entre los métodos y el dominio. Describiremos cómo el propio lenguaje de descripción de métodos, el uso de la clasificación y de la terminología especializada sobre CBR incluida en CBROnto hace que el proceso de *mapping* entre dominio y métodos resulte un proceso intuitivo y simple. Nuestra aproximación se relaciona con uno de los tipos de adaptadores propuestos en UPML: los puentes, y en concreto los que modelan explícitamente la relación entre el dominio y los PSMs.

5. Formalismos de representación de conocimiento: las Lógicas Descriptivas

Sea cual sea el tipo de conocimiento utilizado en un KBS, y en particular el conocimiento de las ontologías, tiene que ser representado utilizando algún formalismo que permita su manipulación. A lo largo de los años, se han definido y utilizado diversos formalismos de representación de conocimiento [Rich&Knight91]: la lógica de predicados, las reglas, las estructuras débiles de ranura y relleno (*slot and filler*) como las redes semánticas y los sistemas de marcos, y las estructuras fuertes de ranura y relleno como los guiones y las estructuras de dependencia conceptual. Nuestra investigación en los últimos años se ha centrado en el uso de las lógicas descriptivas como un formalismo bien fundamentado teóricamente y en las ventajas que sus mecanismos de razonamiento semántico reportan a los sistemas KI-CBR, en concreto a sistemas que requieren una estructura compleja de representación de casos y procesos sofisticados de razonamiento [Gómez *et al.* 99] [González *et al.* 99b] [Díaz&González00a] [Díaz&González01b].

Aunque los sistemas de marcos constituyen un formalismo muy habitual para la representación estructurada de conocimiento, una de sus mayores limitaciones es la falta de una base formal que defina la semántica del conocimiento representado. Recogiendo las ideas básicas de este formalismo e intentando solventar esa carencia de base formal nacen las *Lógicas Descriptivas*. Así pues, las DLs, también denominadas lógicas terminológicas o sistemas terminológicos de representación constituyen un lenguaje de representación de conocimiento que incorpora mecanismos capaces de razonar con la información representada. Las siguientes características resumen sus ideas básicas:

- Enfoque de representación de conocimiento "centrado en los objetos" que permite el modelado de un dominio de aplicación en términos de los objetos o individuos, sus relaciones y las clases de objetos (conceptos):
  - Los conceptos son descripciones de clases de individuos con las que es posible razonar. Los conceptos corresponden a predicados de un argumento que pueden ser aplicados a un individuo y que se describen usando un conjunto de operadores lógicos que depende de la expresividad de la lógica descriptiva utilizada. En el Apartado 5.2.1 se describen varios lenguajes con distinta expresividad.
  - Las relaciones (o roles) son términos formales simples para establecer propiedades. Las relaciones corresponden a predicados de varios argumentos que se utilizan para relacionar individuos.
  - Los individuos son construcciones simples que representan directamente a los objetos del dominio. Las propiedades de los individuos se establecen mediante asertos que indican que satisfacen ciertos conceptos y que se relacionan con otros individuos.
- Definición de un lenguaje de descripción con una semántica bien definida basada en la lógica de predicados de primer orden. Esta característica es lo que diferencia a las DLs de otras representaciones centradas en los objetos. Realmente, las DLs son la base en la que se fundamentan otros formalismos de representación de conocimiento basados en objetos [Calvanese *et al.* 01].
- A diferencia de otros lenguajes basados en objetos, cuando se utilizan DLs se pueden especificar, además de las condiciones necesarias que deben satisfacer los objetos de una clase, las condiciones suficientes que debe cumplir el objeto para pertenecer a esta clase. Esta característica introduce la posibilidad tanto de clasificar automáticamente los objetos en torno a las clases a las que pertenecen (reconocimiento de instancias), como de organizar automáticamente las clases en una jerarquía de subsunción, ya que la definición de una clase permite inferir si es subsumida, subsume o es equivalente a otra clase. Los mecanismos de razonamiento –deducciones lógicas– sobre las descripciones de conceptos y relaciones se pueden clasificar en:
  - Deducciones que completan (añaden restricciones a) las descripciones de los conceptos y las aserciones de los individuos: la herencia de propiedades entre conceptos (relaciones); la compleción de instancias; la combinación de restricciones sobre conceptos (relaciones) e individuos; la propagación de las consecuencias de los asertos; y la detección de inconsistencias.
  - Deducciones lógicas que permiten encontrar la ubicación adecuada de conceptos e individuos dentro del conjunto del conocimiento representado. Es decir, la clasifi-

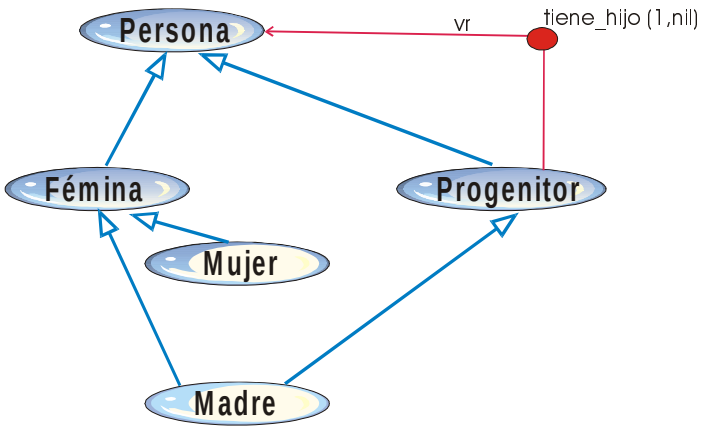


Figura 3-4. *Un ejemplo de representación de conocimiento*

cación de conceptos en una jerarquía de conceptos; y el reconocimiento automático de los conceptos de los que es instancia un individuo dado.

El apartado siguiente introduce algunos conceptos básicos generales de las DLs. El Apartado 5.2 presenta distintos lenguajes de DLs prestando especial atención a su sintaxis básica y semántica. El Apartado 5.3 se encarga de los mecanismos de razonamiento asociados a las DLs y el Apartado 5.4 hace un repaso de la evolución de la investigación en DLs, describiendo las líneas actuales de investigación y haciendo hincapié en los aspectos que determinan su aplicabilidad en distintos ámbitos. El Apéndice A recoge algunas características concretas y la sintaxis de LOOM, el sistema de DLs que hemos utilizado en nuestro trabajo.

5.1 Conceptos básicos

En una típica jerarquía de clases con herencia se representan nodos para caracterizar los conceptos, es decir, las clases de individuos, y enlaces entre nodos para caracterizar las relaciones entre las clases (Figura 3-4). Los conceptos pueden tener propiedades simples llamadas atributos (no se muestran en la figura). De momento obviaremos los detalles sobre los individuos concretos de estas clases. La estructura de figura muestra un ejemplo sencillo para representar conocimiento sobre personas, padres e hijos, a través de las relaciones de generalidad entre los conceptos, comúnmente conocidas como relaciones "es un". Por ejemplo, el enlace entre `Madre` y `Progenitor` en la figura representa que "*las madres son progenitores*".

Además de las relaciones "es un" una característica de las DLs es su capacidad para representar otros tipos de relaciones. En el ejemplo de la figura, el concepto `Padre` tiene una propiedad que se representa mediante un enlace desde el concepto con el nombre `tiene_hijo`. Este tipo de propiedades se llaman roles, relaciones o atributos relacionales (vs. simples). El role `tiene_hijo` tiene una restricción de valor, denotada por la etiqueta `vr`, que significa una limitación en el rango de los tipos de objetos que pueden rellenar este role. Además, el role tiene una restricción de cardinalidad `(1,nil)` en la que el primer número representa la cota inferior del número de hijos y el segundo elemento es la cota superior que no está limitada. La representación del concepto `Progenitor` en la figura se puede leer como: "*un progenitor es una persona que tiene al menos un hijo y todos sus hijos son personas*".

La relación "es un" define una jerarquía sobre los conceptos y proporciona la base para la herencia de propiedades: un concepto más específico que otro hereda las propiedades del

más general. El concepto `Madre` hereda el role `tiene_hijo` –y sus restricciones– de su superconcepto `Progenitor`. Además de los atributos relacionales también se heredan las propiedades representadas por atributos simples, por ejemplo si una `Persona` tiene un atributo `edad`, entonces una `Madre` también tendrá un atributo `edad` (porque también es una persona).

Las ideas anteriores han sido aplicadas en numerosos sistemas de representación, como las redes semánticas, los sistemas de marcos, los guiones, las estructuras de dependencia conceptual o los lenguajes de programación orientados a objetos. De su extenso uso surgió la necesidad de caracterizar de forma precisa el significado de las estructuras utilizadas para la representación así como de las inferencias que se pueden derivar de estas estructuras. Para ello se define un lenguaje para los elementos de la estructura junto con una interpretación adecuada para los constructores del lenguaje.

Para la sintaxis utilizaremos un lenguaje abstracto, similar a otros formalismos lógicos, y dos alfabetos disjuntos de símbolos que se utilizarán para denotar a los *conceptos atómicos (CN)*, representados por símbolos de predicado unario, y a los *roles atómicos (RN)*, representados por símbolos de predicado binarios y que se utilizan para expresar relaciones entre conceptos. Los términos se construyen a partir de los símbolos básicos utilizando distintos tipos de constructores que varían dependiendo de la Lógica Descriptiva concreta. Por ejemplo, la intersección de conceptos, que se denota como ($C \cap D$) o como (and C D), restringe el conjunto de individuos a considerar a aquellos que pertenezcan tanto a C como a D. En la sintaxis de DLs, las expresiones que denotan conceptos no tienen variables. De hecho, una expresión de concepto denota el conjunto de todos los individuos que satisfacen las propiedades especificadas en la expresión. Así, $C \cap D$ puede verse como la sentencia en lógica de primer orden, $C(x) \wedge D(x)$, donde la variable x toma valores en el conjunto de individuos en el dominio de interpretación y C(x) se hace cierto para todos los individuos que pertenecen al concepto C.

Las DLs también se caracterizan por ciertos constructores para establecer relaciones entre los conceptos. Las más básicas son las *restricciones de valor*, escritas $\forall R.C$, que requieren que todos los individuos que están en relación R con un individuo del concepto que estamos describiendo, pertenecen al concepto C.

Respecto a la semántica utilizaremos una interpretación en la que los conceptos se interpretan como conjuntos de individuos y las relaciones se interpretan como conjuntos de pares de individuos. El dominio de interpretación puede ser cualquiera y puede ser infinito. Los conceptos atómicos se interpretan como subconjuntos del dominio de interpretación y la semántica de los distintos constructores se especifica para cada uno de ellos. Por ejemplo, el concepto ($C \cap D$) es el conjunto de individuos que se obtiene de la intersección de los conjuntos de individuos denotados por C y por D. De la misma forma la interpretación de $\forall R.C$ es el conjunto de individuos que están en relación R con individuos que pertenecen al conjunto denotado por el concepto C.

En el ejemplo, supongamos que `Persona`, `Hombre` y `Mujer` son conceptos atómicos y que `tiene_hijo` y `tiene_pariente_femenino` son roles atómicos. Usando los operadores de intersección, unión y negación de conceptos, interpretados como operaciones sobre conjuntos, podemos describir los conceptos "*personas que no son mujeres*" *y "individuos que son hombres o mujeres*" mediante las expresiones `Persona` $\cap \neg$ `Mujer` y `Mujer` $\cup$ `Hombre`, respectivamente. Normalmente se utiliza la terminología lógica y no la de conjuntos, es decir, nos referiremos a la conjunción, disyunción y negación de conceptos.

Entre las *restricciones de cardinalidad* en los roles, la mayoría de las DLs proporcionan cuantificaciones y restricciones de valor asociadas, que permiten por ejemplo la descripción del

concepto "*individuos que tienen una hija*" como ∃tiene_hijo.Fémina, y el concepto "*individuos que sólo tienen hijas (y no hijos)*" como ∀tiene_hijo.Fémina. Los individuos que pertenecen al concepto Fémina son los rellenos del role tiene_hijo.

Las restricciones de cardinalidad permiten caracterizar las relaciones entre los conceptos. De hecho el role entre Progenitor y Persona de la figura puede expresarse como: ∃tiene_hijo.Persona ∩ ∀tiene_hijo.Persona. Esta expresión caracteriza al concepto Progenitor como el conjunto de individuos que tienen al menos un relleno del role tiene_hijo que pertenece al concepto Persona. Es más, todos los rellenos del role tiene_hijo pertenecen al concepto Persona.

Se puede observar que en las restricciones de cuantificación no hay una mención explícita de la variable que se cuantifica. La sentencia equivalente el lógica de primer orden es ∀y.R(x,y) ⊃ C(y), siendo x una variable libre cuyo rango es el dominio de interpretación.

Otro tipo importante de restricciones en los roles son las restricciones de cardinalidad *numéricas*, que restringen la cardinalidad de los conjuntos de rellenos de los roles. Por ejemplo, el concepto de los "*individuos que tienen al menos tres hijos o hijas y al menos dos parientes femeninas*" se expresa como: (≥3 tiene_hijo) ∩ (≤2 tiene_pariente_femenino).

Después de la motivación de las ideas básicas, el siguiente apartado se encarga de una descripción más detallada de los constructores asociados a distintos lenguajes de DLs.

5.2 Lenguajes de descripción

De la introducción anterior podemos destacar los tres puntos siguientes que resumen las ideas básicas de las DLs:

- Los bloques constructivos básicos son los conceptos atómicos (predicados unarios), roles atómicos (predicados binarios) e individuos (constantes).
- La capacidad expresiva del lenguaje –la lógica descriptiva– está restringida al uso de un conjunto de constructores para construir conceptos y roles complejos.
- Los mecanismos de razonamiento permiten inferir conocimiento implícito a partir del conocimiento explícitamente representado.

De los mecanismos de razonamiento nos ocupamos en el Apartado 5.3. En este apartado nos centramos en los dos primeros puntos, haciendo hincapié en el segundo para definir distintas DLs basadas en el uso de distintos conjuntos de constructores.

En una base de conocimiento representada en una DL se distinguen dos componentes (Figura 3-5): la componente terminológica (TBox) contiene conocimiento sobre la terminología o vocabulario de un dominio y la componenete asertiva (ABox) que contiene asertos sobre los individuos del dominio usando el vocabulario de la TBox. El conocimiento terminológico corresponde a las descripciones de las propiedades generales de los términos, conceptos y relaciones, mientras que el conocimiento asertivo da cuenta de la información acerca de los individuos concretos (instancias de los conceptos) del universo de discurso. El conocimiento terminológico (o intensional) se considera más o menos estable en el tiempo, mientras que el conocimiento asertivo (o extensional) es contingente y sujeto a cambios.

5.2.1 La componente terminológica

La componente terminológica -TBox- contiene las clases de individuos, o conceptos, y los roles (relaciones binarias) que es posible establecer entre los individuos. Debido a la naturale-

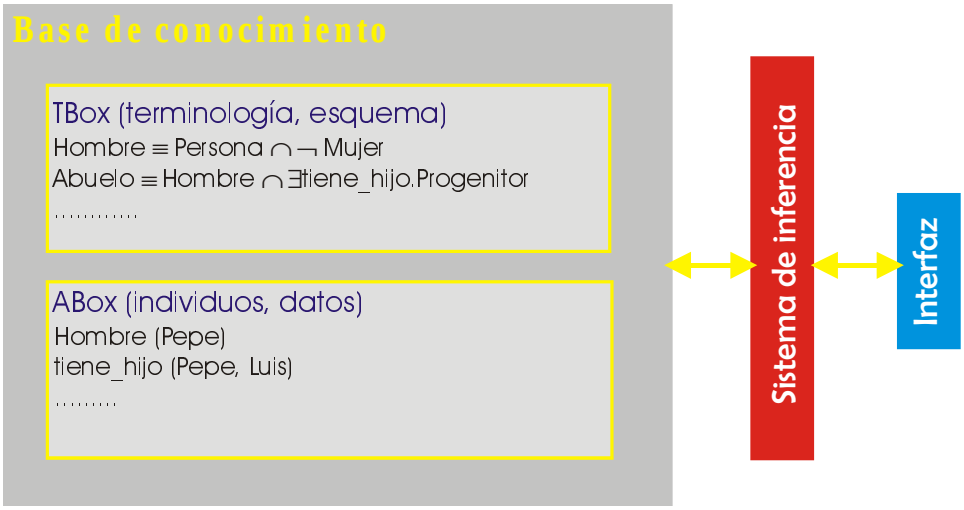


Figura 3-5. *Arquitectura de un sistema de DLs*

za de las relaciones de subsunción entre los conceptos que constituyen la terminología, la TBox tiene una estructura reticular basada en la relación de subsunción (aunque esta estructura no tiene que ver con la implementación real de la subsunción).

Además de los conceptos y roles atómicos, todos los sistemas de DLs permiten a sus usuarios la definición de descripciones complejas de conceptos y algunos sistemas también de relaciones. Además se pueden asignar nombres a estas descripciones complejas. La forma básica de declaración dentro de la TBox es la definición conceptos (relaciones) complejos en base a otros conceptos (relaciones) que se hayan definido previamente. Por ejemplo, el concepto `Mujer` puede definirse como "*una persona de género femenino*" escribiendo: `Mujer ≡ Persona ∩ Fémina.`

Las descripciones de conceptos y relaciones se construyen mediante el lenguaje terminológico. Como ya hemos indicado, las DLs pueden diferir en la expresividad del lenguaje. Dependiendo de las constructoras incluidas, el lenguaje será más o menos expresivo.

El lenguaje menos expresivo de todos es $\mathcal{FL}^-$ (*Frame Language*) [Donini *et al.* 96] que se pueden extender añadiendo otras constructoras. En los siguientes apartados describimos algunos lenguajes de la familia $\mathcal{AL}$ (*Attributive Language*), la primera extensión de $\mathcal{FL}^-$ que resulta de interés práctico.

5.2.1.1 El lenguaje básico $\mathcal{AL}$

Las descripciones de conceptos en $\mathcal{AL}$ se forman según las siguientes reglas sintácticas, donde utilizamos las letras A y B para los conceptos atómicos, las letras R y S para los roles atómicos y las letras C y D para las descripciones arbitrarias de conceptos:

C, D	→	A |	concepto atómico
		⊥ |	concepto bottom
		⊤ |	concepto universal
		¬A |	negación atómica
		C ∩ D |	intersección

$\forall R.\ C$ |	restricción de valor
$\exists R.\top$	cuantificación existencial limitada

Se observa que la negación sólo se puede aplicar a los conceptos atómicos y solo el concepto universal puede utilizarse como ámbito de la cuantificación existencial sobre un role. El lenguaje obtenido al eliminar la negación atómica es $\mathcal{FL}^-$.

Por ejemplo, si suponemos que `Persona` y `Fémina` son conceptos atómicos y que `tiene_hijo` es un role atómico, los siguientes son conceptos válidos definidos con $\mathcal{AL}$: `Persona` $\cap$ `Fémina` (personas de género femenino), `Persona` $\cap \neg$`Fémina` (personas que no son de género femenino), `Persona` $\cap \exists$`tiene_hijo`.$\top$ (personas que tienen un hijo), `Persona` $\cap$ $\forall$`tiene_hijo`. `Fémina` (personas que todos sus hijos son de género femenino, es decir, que sólo tienen hijas), `Persona` $\cap \forall$`tiene_hijo`. $\top$ (personas sin hijos).

La siguiente tabla muestra la semántica denotacional de las descripciones. Para ello se utiliza la noción de *interpretación* $I = < \Delta^I, \cdot^I >$, que está formada por un dominio (universo novacío) de valores Δ^I, y una función $\cdot^I$ que asocia las descripciones de conceptos con *subconjuntos de elementos* del dominio, y las descripciones de relaciones con *conjuntos de tuplas* de elementos del dominio. La *interpretación de un concepto* es el conjunto de todos los individuos del dominio que satisfacen la descripción del concepto. La *interpretación de una relación* es el conjunto de pares de individuos relacionados mediante dicha relación.

$\top^I$	Δ^I (todos los individuos del dominio)
$\bot^I$	$\varnothing$
A^I	$A^I \subseteq \Delta^I$
$(\neg A)^I$	$\Delta^I \setminus A^I$
$(C \cap D)^I$	$C^I \cap D^I$
$(\forall R.\ C)^I$	$\{a \in \Delta^I \mid \forall b.\ (a,b) \in R^I \rightarrow b \in C^I\}$
$(\exists R.\top)^I$	$\{a \in \Delta^I \mid \exists b.\ (a,b) \in R^I\}$

5.2.1.2 La familia de lenguajes que extienden $\mathcal{AL}$

A partir del lenguaje $\mathcal{AL}$ podemos obtener lenguajes más expresivos añadiendo constructores adicionales:

- La unión de conceptos (representada por la letra $\mathcal{U}$) se escribe como $C \cup D$ y se interpreta como: $(C \cup D)^I = C^I \cup D^I$
- La cuantificación existencial completa (representada por la letra $\mathcal{E}$) se escribe como $\exists R.C$ y se interpreta como: $(\exists R.C)^I = \{a \in \Delta^I \mid \exists b.\ (a,b) \in R^I \wedge b \in C\}$. La diferencia con la cuantificación existencial limitada $\exists R.\top$ es que cualquier concepto puede aparecer en el ámbito del cuantificador existencial.
- Las cuantificaciones numéricas o de cardinalidad (representadas por la letra $\mathcal{N}$) se escriben como $\geq$ n.R (restricción de cardinalidad mínima) y $\leq$ n.R (restricción de cardinalidad máxima), donde el rango de n son los enteros no negativos. Se interpretan como:

  $(\geq n.R)^I = \{a \in \Delta^I \mid \ |\{b \mid (a,b) \in R^I\}| \geq n\}$

  $(\leq n.R)^I = \{a \in \Delta^I \mid \ |\{b \mid (a,b) \in R^I\}| \leq n\}$

donde | . | representa la cardinalidad de un conjunto.

- La negación de conceptos arbitrarios (representada por la letra $\mathcal{C}$) se escribe como ¬C y se interpreta como: $(\neg C)^I = \Delta^I \setminus C^I$

Con los constructores adicionales se podría describir, por ejemplo, el concepto: "*personas que o bien tienen como mucho un hijo o bien tienen más de tres hijos uno de los cuáles es niña*":

Persona∩(≥1.tiene_hijo ∪(≤ 3.tiene_hijo ∩ ∃tiene_hijo.Fémina))

La extensión del lenguaje $\mathcal{AL}$ con cualquier subconjunto de estos constructores produce un lenguaje de la familia $\mathcal{AL}$. Se usa el convenio de denotar a los lenguajes por cualquier cadena de caracteres de la forma: $\mathcal{AL}$ [$\mathcal{U}$] [$\mathcal{E}$] [$\mathcal{N}$] [$\mathcal{C}$], donde la presencia de una letra representa la aparición del constructor correspondiente. Por ejemplo, $\mathcal{ALU}$ es la extensión de $\mathcal{AL}$ que incluye la unión de conceptos; $\mathcal{ALE}$ añade la cuantificación existencial completa de relaciones, $\mathcal{ALEN}$ es la extensión de $\mathcal{AL}$ con cuantificación existencial completa para las relaciones y restricción de cardinalidad en las relaciones, etc.

Desde el punto de vista semántico no todos los lenguajes anteriores son distintos, ya que existen equivalencias, por ejemplo, C ∪ D = (¬C ∩ ¬D) o ∃R.C = ¬ ∀R. ¬C. Por esta razón, la unión y la cuantificación existencial completa se pueden expresar en base a la negación. De la misma forma, la combinación de la unión y la cuantificación existencial completa nos da la posibilidad de expresar la negación de conceptos [Baader *et al.* 02 (Cap 2)]. En [Borgida96] se puede encontrar un estudio y una comparativa profundos de la expresividad de las DLs y de la lógica de predicados.

5.2.1.3 Los axiomas terminológicos

Los lenguajes de descripción permiten construir descripciones complejas de conceptos y relaciones. Para definir una terminología Tbox, se utilizan estas descripciones en los *axiomas terminológicos*, que se utilizan para definir cómo los conceptos (y los roles) se relacionan entre sí. Una terminología se define como un conjunto de definiciones (axiomas terminológicos) según las cuáles podemos introducir conceptos (o roles) atómicos como abreviaturas (o nombres) de otros conceptos (o roles) más complejos. La descripción de un concepto contiene las restricciones que deben verificar los individuos que pertenecen a dicha clase de entidades, esto es, los que son instancias del concepto. La descripción de una relación contiene las restricciones que deben cumplir los individuos entre los cuales se establece.

Existen dos suposiciones básicas sobre la Tbox. Cada nombre de término se define una única vez y las definiciones no son cíclicas en el sentido de que ningún término se define en base a sí mismo o a cualquier término que indirectamente se refiera a él (aunque hay extensiones de las DLs que sí permiten ciclos). Para simplificar la exposición nos centraremos en los conceptos ya que no todos los sistemas permiten definir roles complejos. En el Apéndice A incluimos el lenguaje de descripción de relaciones de LOOM.

Los axiomas terminológicos restringen el conjunto de interpretaciones a tener en cuenta. Existen dos tipos de axiomas:

- Axiomas de definición que proporcionan definiciones para algunos nombres de concepto de la forma A ≡ D o A ⊆ D donde A es un concepto atómico (A ∈ *CN*) y D es una descripción arbitraria. Si estas definiciones no producen ciclos (es decir, no hay recursión), sólo son abreviaturas que pueden ser expandidas.

  Por ejemplo: Padre ≡ Hombre ∩ ∃tiene_hijo.Persona

$$\texttt{Humano} \subseteq \texttt{Animal} \cap \text{Bípedo}.$$

- Axiomas de inclusión de la forma D ⊆ E para descripciones arbitrarias D y E, que permiten asertar relaciones de subsunción. Por ejemplo: $\exists\texttt{tiene\_hijo.Varón} \subseteq \exists\texttt{tiene\_hijo.Persona}$

El significado de una descripción D de una TBox sólo tiene en cuenta las interpretaciones en las que se cumplen los axiomas terminológicos de la TBox. Los axiomas de definición se interpretan de la siguiente forma:

- A ≡ D corresponde a la definición de A como un concepto *definido* y se interpreta como una equivalencia lógica que proporciona las condiciones necesarias y suficientes para clasificar a un individuo como una instancia del concepto A. El hecho de que sean condiciones suficientes de pertenencia al concepto (de capacidad de "ser instancia del" concepto) hace que, si el conjunto de hechos conocidos acerca de un individuo verifica las restricciones de la descripción, el sistema, gracias a sus mecanismos de razonamiento, infiera de modo automático que el individuo pertenece al concepto. Por su parte, el hecho de que las condiciones de la descripción sean condiciones necesarias hace que, si se aserta que un individuo es una instancia de un concepto, automáticamente el individuo pase a verificar todas las restricciones que aparecen en su definición.

- A ⊆ D corresponde a la definición de A como un concepto *primitivo*, utilizando únicamente condiciones necesarias pero *no suficientes* de pertenencia al concepto A. Cumplir la descripción dada en D es una condición necesaria para que un individuo sea instancia del concepto A; sin embargo no es suficiente para inferir la pertenencia de un individuo al concepto A. Esto significa, en ningún caso se puede inferir automáticamente que un individuo es una instancia de un concepto primitivo. Pero si se aserta explícitamente que un individuo es instancia de un concepto primitivo, el sistema aplicará todas las restricciones del concepto. En cierto modo, se pueden ver las descripciones de los conceptos primitivos como descripciones incompletas, incluso se puede definir un concepto primitivo sin asociarle descripción alguna, simplemente con el objetivo de introducir un nuevo término en la representación. De ahí que el sistema no pueda realizar inferencias automáticas en estos casos.

El hecho de que al modelar el dominio una clase se represente como un concepto primitivo o como un concepto definido *no es una propiedad intrínseca de dicha clase* sino que es una decisión que se toma al modelar el dominio. Esta decisión se basa en la granularidad de la representación y el tipo de inferencias que se pretendan conseguir. Así, por ejemplo, en un modelo la clase *persona* se puede representar como un concepto primitivo mientras que en otro se puede considerar como un concepto definido si cualquier "animal bípedo e implume" se debe reconocer como una persona [Brachmann *et al.* 91].

Las relaciones, por su parte, al igual que los conceptos, también pueden ser primitivas o definidas. Las primitivas introducen condiciones necesarias para los individuos entre los que se establecen y las definidas introducen condiciones necesarias y suficientes. Estas últimas permiten, por lo tanto, identificar una tupla ordenada de individuos como instancia de dicha relación.

5.2.2 La componente asertiva

La segunda componente de una base de conocimiento es la componente asertiva -ABox- que contiene conocimiento extensional sobre el dominio de interés y representa una particularización total o parcial del esquema genérico representado por la TBox. Contiene las aserciones o hechos que relacionan a los individuos con los conceptos o que relacionan individuos entre sí. Denotando a los individuos con las letras a, b, c, podemos realizar asertos de dos tipos:

- Pertenencia a un concepto: `C(a)` que significa que `a` pertenece a (la interpretación de) `C`.
- Relación con otro individuo: `R(b,c)` que significa que `c` es un relleno del role `R` del individuo `b`.

Por ejemplo, si `Ana` es un nombre de individuo entonces `Fémina`$\cap$`Persona (Ana)` establece que `Ana` es una persona de género femenino. Suponiendo la definición anterior del concepto `Mujer` (`Mujer` $\equiv$ `Persona` $\cap$ `Fémina`) uno puede inferir del aserto anterior que `Ana` es una instancia del concepto `Mujer`.

De forma similar la expresión `tiene_hijo (Ana, Luis)` especifica que `Ana` tiene como hijo al individuo `Luis`.

La semántica de la Abox se obtiene extendiendo la interpretación a los nombres de individuos, de forma que a cada nombre de individuo `a` le corresponde un elemento del dominio de interpretación: $a^I \in \Delta^I$. Esta interpretación debe respetar la identidad única de los individuos, es decir, si `a` y `b` son nombres distintos entonces $a^I \neq b^I$. Es decir, individuos con nombres distintos representan entidades distintas aún cuando sus descripciones sean exactamente iguales.

En general, e independientemente del grado de expresividad que tenga una DL, la expresividad de su lenguaje asertivo suele ser más reducida que la de su lenguaje terminológico. La razón: conseguir un rendimiento satisfactorio. La ABox de KRIPTON [Brachman *et al.* 85], uno de los sistemas terminológicos de representación, establecía una correspondencia completa con el cálculo de predicados de primer orden y eso hacía necesario un demostrador de teoremas basado en resolución para realizar las inferencias. El resultado: un sistema indecidible que además resulta demasiado lento y frágil para poder ser utilizado en aplicaciones interactivas [MacGregor91].

El intento de obtener componentes asertivas eficientes se debe a suposiciones que se hacen en el diseño general de los sistemas terminológicos de representación. Podemos resumirlas en dos. Por un lado se considera que la componente terminológica es más estable, –esto es, evoluciona menos y se considera básicamente estática– que la componente asertiva, sujeta a continuas evoluciones de los asertos a medida que vamos ampliando el conocimiento del mundo modelado. Por otro lado, el tamaño de la componente asertiva se considera mucho mayor que el de la terminológica, ya que resulta razonable suponer que el número de individuos sea mayor, con diferencia, que el número de clases de individuos.

5.3 Mecanismos de inferencia de las Lógicas Descriptivas

Además de la semántica formal basada en lógica, la otra característica fundamental de las DLs es el énfasis en los mecanismos de razonamiento que permiten inferir conocimiento que

está implícito a partir del conocimiento que explícitamente se representa en una base de conocimiento.

El mecanismo básico de inferencia sobre las expresiones que representan conceptos en un sistema de DLs es la *subsunción*. Pero además, existen otros mecanismos de inferencia asociados a las definiciones conceptuales como la *satisfactibilidad* de un concepto, la *equivalencia* entre dos expresiones conceptuales o la comprobación de si dos conceptos son *disjuntos*. Respecto a la ABox, la principal tarea de razonamiento es la comprobación de que un cierto individuo pertenece, es decir, es una instancia de un cierto concepto. Aunque también se emplean otros mecanismos se pueden definir en base al anterior. Por ejemplo, la comprobación de la *consistencia* de una base de conocimiento verifica si cada concepto de la misma admite al menos un individuo; el *reconocimiento de instancias* encuentra el concepto más específico del que un cierto individuo es instancia y la *recuperación* que encuentra todos los individuos de la base de conocimiento que son instancias de un concepto dado.

El tipo de procesos de razonamiento de los que se ocupa la comunidad de DLs son procedimientos de decisión, que a diferencia de otros tipos, por ejemplo, demostradores automáticos de teoremas, deben terminar tanto para respuestas positivas como negativas. Sin embargo, poder garantizar una respuesta en tiempo finito no implica, en absoluto, que la respuesta sea obtenida en un tiempo razonable, por lo que la investigación acerca de la complejidad computacional de una cierta DL con procesos de inferencia decidibles es un aspecto importante. La decidibilidad y complejidad de los mecanismos de inferencia dependen directamente de la capacidad expresiva de la DL considerada. Por un lado, las DLs muy expresivas tienen asociados procesos de inferencia de complejidad muy elevada, e incluso indecidible. Por el otro lado, las DLs poco expresivas con procesos de razonamiento muy eficientes pueden no ser suficientemente expresivas para las necesidades de representación de una aplicación. Por esto, como veremos en el Apartado 5.4, la investigación del compromiso entre la expresividad de las DLs y la complejidad de sus mecanismos de razonamiento ha sido uno de las áreas más activas de investigación en la comunidad de DLs.

5.3.1 Subsunción y clasificación

Todos los sistemas basados en DLs incluyen un mecanismo de razonamiento por el que son capaces de, dadas dos descripciones de conceptos, determinar si una descripción implica o es *subsumida* por otra[21]. Esa implicación se interpreta de la siguiente forma: la descripción de un concepto D implica la del concepto C cuando cualquier individuo que satisface la descripción de D, esto es, es instancia de D, debe satisfacer la descripción de C. En consecuencia, el conjunto de instancias de D es un subconjunto del conjunto de las instancias de C. Cuando lo anterior ocurre, decimos que el concepto C subsume al concepto D, que D es subsumido por C, que C es un superconcepto o generalización de D o que D es un subconcepto o una especialización de C, y lo representamos como $C \subseteq D$ [Borgida92] [Woods91].

Determinar la subsunción es el problema de comprobar si el concepto D (el subsumidor) es más general que el concepto C (el subsumido) para cualquier interpretación I que satisfaga los axiomas de la TBox: $D^I \subseteq E^I$.

La relación de subsunción está relacionada con los enlaces "es un" que aparecen en otros formalismos de representación de conocimiento, por ejemplo, en las redes semánticas, donde dichos enlaces son explícitamente introducidos por el usuario. La diferencia fundamental es que en las DLs las relaciones de subsunción entre conceptos y el reconocimiento de las ins-

[21] Nos restringimos a la subsunción entre conceptos. La subsunción entre relaciones sólo se incluye en DLs muy expresivas.

tancias que pertenecen a un concepto pueden ser inferidas a partir de la definición de los conceptos y de las propiedades de los individuos.

Existen básicamente dos aproximaciones para determinar la relación de subsunción entre dos conceptos y, por tanto, para resolver todos los procesos de razonamiento de las DLs, ya que todos ellos se basan en la relación de subsunción. Algunas implementaciones de sistemas de DLs utilizan técnicas de demostración de teoremas por medio de *tableaux* y reglas de reescritura. Otras usan lo que se denomina "comparación estructural" entre descripciones. En [Woods91] se puede encontrar un análisis completo de cómo se determina la relación de subsunción utilizando estos criterios estructurales. Aunque no entraremos en detalle remitimos al lector interesado a [Baader *et al.* 02].

De manera informal, las reglas que utiliza la aproximación estructural para decidir que una descripción *X* subsume a otra, *Y*, son:

- Un concepto que aparece en la descripción *X* es más general que un concepto que aparece en *Y*. Por ejemplo, "persona cuyos hijos son médicos", subsume a "mujer cuyos hijos son médicos", porque `Persona` es más general que `Mujer`.
- Las restricciones sobre una relación en *X* son más generales que las restricciones sobre esa misma relación en *Y*. Por ejemplo, "una persona cuyos hijos son profesionales" subsume a "persona cuyos hijos son médicos" porque la restricción "hijos profesionales" es más general que "hijos médicos".
- La descripción más general, *X*, no incluye un concepto, o una restricción, que sí aparece en la descripción más específica, *Y*. Por ejemplo, "una persona cuyos hijos son médicos" subsume a "una persona cuyos hijos son médicos y que le gusta conducir".

La relación de subsunción introduce un orden parcial entre los conceptos (parcial porque puede ocurrir que dos conceptos no sean comparables). Esta relación de orden parcial junto con un concepto genérico que, por definición, subsume a cualquier otro, y que se encuentra en todas las DLs, permite organizar un grupo de descripciones en una jerarquía de especialización, en la que el concepto genérico se encuentra en la raíz y los conceptos más específicos (los que no subsumen a ningún otro) se encuentran en las hojas.

La misma capacidad que permite determinar si un concepto subsume a otro es la que permite a las DLs clasificar automáticamente un concepto con respecto a una jerarquía de conceptos. Esto es, capacita para buscar la posición que ocupa el nuevo concepto en la jerarquía de especializaciones, de manera que los conceptos a los que el nuevo subsume queden por debajo de él en la nueva jerarquía y los conceptos que le subsumen queden por encima. La componente del sistema encargada de proporcionar esta funcionalidad es el *clasificador* y es una de las características esenciales de las DLs.

Otro ejemplo típico de inferencia basado en la subsunción es el problema de comprobar la *satisfactibilidad* de un concepto. La satisfactibilidad es un caso particular de subsunción, donde el subsumidor es el concepto vacío, que significa que un concepto no es satisfactible. Una descripción de un concepto D es coherente/satisfactible, si existe al menos una interpretación I que satisfaga los axiomas de la Tbox, tal que $D^I \neq \varnothing$. El sistema se encarga de la detección de conceptos *incoherentes* (incoherencias que aparecen explícitamente en la definición o después de combinar las restricciones locales con las heredadas).

También se puede comprobar la *equivalencia* entre conceptos. Dos conceptos C y D son equivalentes si y sólo si $C^I = D^I$ para cualquier interpretación I que satisfaga los axiomas de la Tbox. O si dos conceptos son *disjuntos*. Dos descripciones de conceptos D y E son disjuntas si y sólo si $D^I \cap E^I = \varnothing$ para cada interpretación I que satisfaga los axiomas de la Tbox .

```
Mujer ≡ Persona ∩ Femina
Hombre ≡ Persona ∩ ¬ Mujer
Madre ≡ Mujer ∩ ∃tiene_hijo.Persona
Padre ≡ Hombre ∩ ∃tiene_hijo.Persona
Progenitor ≡ Padre ∪ Madre
Abuela ≡ Madre ∩ ∃tiene_hijo.Progenitor
Madre_de_familia_numerosa ≡ Madre ∩ ≥3. tiene_hijo
Madre_sin_hijas ≡  Madre ∩ ∀tiene_hijo. ¬ Mujer
Esposa ≡ Mujer ∩ ∃tiene_marido.Hombre
```

Figura 3-6. *Ejemplo de terminología (Tbox) con conceptos sobre las relaciones familiares*

Por ejemplo, con respecto a la terminología de la Figura 3-6, `Persona` subsume a `Mujer`, tanto `Mujer` como `Progenitor` subsumen a `Madre`, y `Madre` subsume a `Abuela`. Además, `Mujer` y `Hombre`, y `Padre` y `Madre` son disjuntos. Las relaciones de subsunción se pueden inferir de las definiciones debido a la semántica de $\cap$ y $\cup$, y que `Hombre` y `Mujer` son disjuntos se infiere del hecho de que `Hombre` sea subsumido por la negación de `Mujer`.

Lo que aquí se ha expuesto para los conceptos también puede, en ocasiones y sólo para lenguajes muy expresivos, aplicarse a las relaciones. En ciertos sistemas, las relaciones también pueden organizarse en una jerarquía de especialización, es decir, en esos sistemas las relaciones son, al igual que los conceptos, consideradas como "ciudadanos de primera clase". Así pues, cuando el conocimiento se representa utilizando DLs pueden existir dos taxonomías: la de conceptos y la de relaciones. Cualquier taxonomía de conceptos será, casi con toda seguridad, mucho más amplia horizontal y verticalmente que la de relaciones, pero ambas pueden resultar de gran interés.

5.3.2 Reconocimiento y compleción de instancias

Un individuo se crea, o ve aumentada las características que lo definen, mediante aserciones que, de manera explícita, permiten establecer que el individuo es instancia de uno o más conceptos y/o que está relacionado con otro(s) a través de una cierta relación. El conjunto de conceptos de los que un individuo es instancia recibe el nombre de *tipo* del individuo.

Llamamos reconocimiento de instancias a la capacidad de razonamiento del sistema para conocer en cada momento el tipo de los individuos. Y denominamos *reconocedor* a la componente del sistema encargada de dicha funcionalidad. La misión del reconocedor es mantener el tipo de los individuos, para lo cual debe estar atento a cada cambio en el conocimiento sobre los individuos y sobre la TBox.

Existen dos enfoques básicos para implementar el reconocedor. Uno de ellos es considerar al individuo como una descripción y clasificarla por medio del clasificador. El otro consiste en introducir explícitamente en el lenguaje la relación "es instancia de". En [Mac Gregor 88] se analiza cómo esta segunda estrategia, aunque menos elegante, proporciona mejores resultados en términos de eficiencia. De ahí que suela ser empleada. Mientras que la clasifica-

ción se puede considerar como una operación que se realiza *off-line*, el reconocimiento de instancias tendrá lugar en gran parte durante la interacción, por lo que resulta fundamental que esta operación esté optimizada, dada la preocupación por la eficiencia de la componente asertiva que ya mencionamos anteriormente.

Al asertarse o inferirse que un individuo es instancia de un cierto concepto, las condiciones necesarias que se incluyen en la definición del concepto pueden provocar determinadas aserciones de forma automática[22]. Se trata de la compleción de instancias, un mecanismo de inferencia que enriquece automáticamente la descripción de un individuo asignando valores específicos a algunas de sus relaciones.

5.3.3 Otros mecanismos de inferencia

Como resultado del proceso de clasificación y la consiguiente organización taxonómica, todas las propiedades de un concepto dado se propagan a todos sus subconceptos. Esto es, los subconceptos *heredan* las propiedades de los conceptos que los subsumen y la clasificación taxonómica nos permite hacer uso de dicho mecanismo de herencia (lo mismo es válido para las relaciones). La herencia de propiedades en la jerarquía de subsunción proporciona los siguientes beneficios:

- Los usuarios pueden inspeccionar los conceptos para ver si las propiedades que se aplican de manera lógica coinciden con sus expectativas, lo cual ayuda en la comprensión de la forma en que los sistemas terminológicos razonan y a la vez permite detectar errores introducidos por el usuario en la construcción de la base de conocimiento.
- El sistema detecta automáticamente conceptos con propiedades conflictivas, propiedades resultado de *combinar* las restricciones que heredan o de combinar las que heredan con las restricciones locales de su definición. Se dispone pues de un mecanismo de *detección de incoherencias* en conceptos. Dichos conceptos, al ser incoherentes, no podrán tener instancias.

Podríamos hablar también de inferencia a nivel de individuos: cuando un nuevo individuo se describe en términos de conceptos existentes, el primero hereda las propiedades de los segundos. De igual manera también podemos hablar de combinar las restricciones impuestas sobre los individuos, las cuales conducen a ciertas conclusiones lógicas. Además, cuando un individuo se crea, la herencia y la combinación de propiedades pueden hacer que cierta información resultado de consecuencias lógicas se *propague* a otros individuos relacionados con él. Durante dicha propagación de propiedades tiene lugar la *detección de incoherencias*, incoherencias que también pueden ser resultado de aserciones realizadas explícitamente acerca de hechos cuya coexistencia es imposible[23].

Otro mecanismo adicional es la aplicación de *reglas* que se representan como condiciones necesarias asociadas a la definición de los conceptos. Cuando se reconoce un individuo como instancia de un concepto, se aplican sobre ese individuo las condiciones necesarias incluidas en la descripción del concepto. De esta forma, los conceptos actúan como antecedentes de las reglas.

La posibilidad de asociar reglas con los conceptos es, en cierto modo, un mecanismo más limitado que las reglas de producción, pues el antecedente de las reglas normalmente es un

[22] Esto es posible si el reconocedor "razona hacia adelante".

[23] Las incoherencias también se detectan tras realizar cada aserto.

único concepto, un predicado unario. En el caso de los conceptos definidos el antecedente es la definición del concepto, ya que el sistema reconoce automáticamente cuándo un individuo satisface la definición y aplica sobre el individuo las condiciones necesarias y las restricciones por defecto que ese concepto lleve asociadas. Sin embargo, el hecho de que los conceptos estén organizados en una jerarquía de especializaciones también supone ciertas ventajas, por ejemplo, que las reglas se organizan automáticamente, con lo cual es fácil encontrar reglas relacionadas. Además, la clasificación es un medio intuitivo de resolver conflictos entre reglas. Cuando en una situación determinada es posible aplicar más de una regla, uno de los criterios de selección más habituales consiste en aplicar la regla más específica, lo que, en un sistema terminológico de representación, se traduce directamente en seleccionar la regla asociada con el concepto más específico.

Otros procesos de inferencia no estándar puede ser útiles para una gran variedad de propósitos, como el soporte para la construcción de bases de conocimiento y la obtención de información sobre el conocimiento representado. Entre los procesos de inferencia no estándar de las DLs se encuentra el cómputo del LCS (*least common subsummer*) o concepto común más específico que subsume a un conjunto de conceptos dados. Es decir, que no existe otro concepto que subsuma a todos los conceptos del conjunto y que sea subsumido por el LCS. Fue definido en el ámbito de DLs en [Cohen *et al.* 92] y ha sido utilizado en diversas áreas: aprendizaje inductivo de descripciones de conceptos a partir de ejemplos, construcción de bases de conocimiento a partir de las instancias de los conceptos (aproximación *bottom-up* en vez de la aproximación clásica *top-down*), o la representación de disyunción en las DLs que no la incluyen. La noción de LCS está muy relacionada con el concepto más específico (MSC) asociado a un individuo, es decir, la descripción conceptual mínima de la que el individuo es instancia, dados los asertos sobre el individuo.

5.4 Evolución y uso de las Lógicas Descriptivas

Como hemos descrito en la introducción, las DLs surgieron de la investigación relativa a los formalismos de representación de conocimiento, más concretamente relacionada con aquellos que comparten la idea de que la estructura del conocimiento se puede expresar en términos de las clases de objetos relevantes en un cierto dominio y de las relaciones entre ellos. Los primeros formalismos con estas características fueron las redes semánticas y los sistemas de marcos que fueron definidos en general de un modo informal, haciendo que las herramientas de razonamiento asociadas a dichos formalismos fueran muy dependientes de las estrategias de implementación inherentes a cada uno.

Un paso fundamental hacia una caracterización basada en lógica de este tipo de sistemas se consiguió con el sistema KL-ONE [Brachman&Schmolze85], que proporcionaba una base lógica para la interpretación de objetos, clases (conceptos) y sus relaciones.

El objetivo básico de esta fundamentación lógica fue caracterizar un conjunto de constructores para la definición de términos (conceptos y relaciones) y dotarlos de un significado formal. Además, un sistema de representación de este tipo debería proporcionar mecanismos de razonamiento correctos y completos respecto a la semántica formal especificada para los constructores de términos, y que se puedan caracterizar en términos de su complejidad computacional. Los artículos [Brachman&Levesque84] [Levesque&Brachman85] se consideran los primeros en DLs y abrieron una línea de investigación de estudio del compromiso entre la expresividad y la complejidad computacional de los mecanismos de razonamiento de distintos conjuntos de constructores terminológicos. De hecho, se mostraba como una extensión aparentemente pequeña del lenguaje puede hacer los procesos deductivos incluso inde-

cidibles. Analizó el lenguaje $\mathcal{FL}^-$ donde la determinación de la subsunción puede ser resuelta en tiempo polinomial. Sin embargo, para el lenguaje $\mathcal{FL}$ que resulta de la inclusión de un constructor de restricción de valor para roles hace que el problema pase a ser co-NP-completo.

Los primeros resultados sobre la complejidad en el caso peor [Brachman&Levesque87] [Nebel90] mostraron que la resolución de la subsunción es intratable, es decir, no se puede resolver en tiempo polinomial, incluso para DLs muy poco expresivas. Este fue el punto de partida en el que varios grupos de investigación han centrado su atención implementado variantes que, aunque desarrolladas bajo el marco común de las DLs, difieren en aspectos como la expresividad del lenguaje de representación, la completitud de los mecanismos de razonamiento, la eficiencia, la interfaz de usuario o la integración con otros modos de razonamiento [MacGregor91] [Brachman *et al.* 85] Donini *et al.* 95] [Donini *et al.* 96].

En [Donini *et al.* 99] se presenta una visión completa de la frontera de la intratabilidad en DLs, haciendo un estudio del compromiso óptimo expresividad/complejidad, en concreto de la extensión más expresiva de $\mathcal{FL}^-$ con respecto a un conjunto de constructores que mantienen polinomial la complejidad de la subsunción.

La intratabilidad del razonamiento asociado a las DLs (en el sentido de no ser polinomial en el caso peor) no impide que una DL sea útil en la práctica, si se utilizan técnicas de optimización al implementar un sistema basado en esa DL, por lo que otra línea de investigación se refiere al estudio de estos aspectos [Baader *et al.* 02 (Cap 9)].

Después de más de una década de estudio, esta línea de investigación se ha dado por completada y la comunidad de DLs tiene una idea global del compromiso entre expresividad y complejidad [Donini *et al.* 99]. El resultado es que salvo los lenguajes con una expresividad muy restringida, el problema de la determinación de la relación de subsunción es co-NP-completo o incluso indecidible. El resultado de todo esto: la necesidad de llegar a un compromiso entre expresividad, eficiencia y completitud.

Distintos sistemas adoptan soluciones muy diferentes. Tenemos sistemas con razonadores bastante completos y relativamente eficientes a cambio de ser muy poco expresivos. El sistema CLASSIC [Brachman *et al.* 91] es el más representativo de este tipo. Otros sistemas ofrecen razonadores completos pero computacionalmente intratables para lenguajes muy expresivos, como es el caso de KRIPTON. Y por último, existen sistemas que proporcionan razonadores eficientes para lenguajes muy expresivos a cambio de sacrificar la completitud. Es el caso de BACK [Nebel&VonLuck88] y LOOM [MacGregor88][MacGregor91] [MacGregor&Bates87]. LOOM es un sistema desarrollado en el *Information Science Institute de la University of Southern California* por el grupo de investigación encabezado por Robert Mac Gregor.

Después de estudiar la fuente de la incompletitud e identificar los constructores –o de forma más precisa, las combinaciones de constructores– que requieren algoritmos exponenciales para preservar la completitud del razonamiento, se desarrollaron sistemas como KRIS [Baader&Hollunder91] que se caracterizan por lenguajes expresivos (relativamente) y con mecanismos de razonamiento completos. Aunque son mucho menos eficientes que el resto de las aproximaciones resultaron muy útiles en estudios comparativos con otros sistemas. Uno de estos estudios comparativos realizado en el año 1994 identifica a LOOM como el más expresivo y eficiente entre seis sistemas terminológicos de representación (BACK, CLASSIC, KRIS, MESON, SB-ONE y LOOM) [Heinsohn *et al.* 94]. El estudio concluye que:

- La eficiencia de los sistemas de representación terminológicos no depende únicamente de los algoritmos de cálculo de la relación de subsunción sino también de los algoritmos de clasificación.

- Cuanto más expresivo y más completo es un sistema, más lento es.
- La estructura de la base de conocimiento tiene un impacto muy importante en el rendimiento, incluso no utilizando ejemplos artificiales (buscando los casos peores), sino bases de conocimiento reales.
- El tiempo de ejecución de los sistemas crece de forma cuadrática con el tamaño de la base de conocimiento.

En [MacGregor91] justifica la aproximación elegida por LOOM de la siguiente manera:

- Los usuarios solicitan lenguajes más y más expresivos con los que describir los dominios a modelar.
- Si los formalismos de representación carecen de los mecanismos de razonamiento que los KBSs necesitan, el desarrollador incluirá dichos mecanismos por su cuenta y riesgo, obteniéndose soluciones inferiores en calidad a las obtenidas si los mecanismos están incluidos en los propios formalismos.
- La complejidad inherente al cálculo de la relación de subsunción es irrelevante si en tiempo de ejecución se trabaja con individuos y recae más peso en el reconocedor.

Las elevadas expresividad y eficiencia, unidas a la diversidad de herramientas incluidas en LOOM son las que nos han hecho elegirlo como sistema de representación de conocimiento, ya que ofrece las facilidades que los desarrolladores necesitan y puede ser utilizado en el desarrollo de aplicaciones de tamaño real. Ahora bien, hay que ser conscientes de que tiene ciertas limitaciones a nivel de completitud. El Apéndice A, describe la sintaxis y, de manera informal, la semántica de los lenguajes de descripción de términos y aserciones de LOOM. Así como el lenguaje de construcción de consultas que acompaña al sistema.

En el siguiente apartado se describe la relación de las DLs con otros formalismos de representación de conocimiento. El Apartado 5.4.2 repasa las características que hacen de las DLs un formalismo idóneo para la representación de ontologías. En el Apartado 5.4.3 se describen las tendencias actuales guiadas por el uso de las DLs en distintos tipos de aplicaciones. El Apartado 5.4.3 hace hincapié en el uso de las DLs en el tipo de aplicaciones en el que se centra esta tesis: los sistemas KI-CBR.

5.4.1 Relación con otros formalismos de representación

Las DLs se relacionan con otros formalismos de representación basados en objetos y comparten las mismas motivaciones que los mecanismos de representación basados en redes o estructuras conceptuales. En [Lehmann92] se revisan varios lenguajes orientados a la representación estructurada de conocimiento y se comparan con las DLs. No incidiremos más en la relación entre las DLs, las redes semánticas y los sistemas de marcos, que ya hemos descrito como motivación del desarrollo de las DLs. Sólo citar que en los marcos se permiten características adicionales como valores por defecto (que son incluidos como extensiones en algunas DLs muy expresivas) y aspectos dinámicos como disparadores. En [Baader *et al.* 99] se lleva a cabo un análisis detallado de las relaciones entre las DLs y los grafos conceptuales [Sowa91]. La conclusión principal es que ya que cualquier DL incluye las restricciones de roles cuantificadas universalmente, y que ésta característica no está presente en los grafos conceptuales, las estructuras de representación se deben interpretar de forma sustancialmente distinta.

En muchos otros campos de la Informática se encuentran formalismos para representar clases y objetos, que comparten la noción de clase como subconjunto del universo de discur-

so, y permiten expresar restricciones (por ejemplo subclases) y relaciones entre clases. En particular estos formalismos se han desarrollado en el ámbito de los sistemas de bases de datos, modelos de datos, lenguajes orientados a objetos y en general, en los lenguajes de programación.

Aunque ha habido diferentes trabajos que se han encargado de la relación entre los distintos formalismos basados en objetos desarrollados en las distintas áreas, y las DLs, es difícil encontrar un marco común que permita hacer una comparación precisa. En [Baader *et al.* 02 (Cap 4)] se utiliza una DL como marco común para identificar las características comunes entre distintos modelos de datos. En particular especifican la correspondencia entre una DL y el modelo Entidad-Relación y un modelo de datos orientado a objetos para concluir que, aunque se pueden identificar muchas similitudes y puntos básicos en común, existen muchas características distintas. Por ejemplo, en algunos modelos de datos se requiere una forma de asertos de inclusión cíclicos, y roles bidireccionales para modelar relaciones en ambos sentidos, mientras que los roles de DLs tiene una direccionalidad bien definida. Respecto a los modelos de datos orientados a objetos la diferencia fundamental es que aunque las DLs proporcionan la expresividad suficiente para modelar estructuras de registros y conjuntos, no forman parte de los mecanismos explícitos que se pueden utilizar, por lo que su representación es artificial y poco simple para quién la utiliza. Lógicamente el punto fuerte de las DLs son sus mecanismos de razonamiento que no se incluyen en ninguno de los modelos de datos. Estos mecanismos resultan de gran utilidad y existen muchos trabajos que estudian las aplicaciones de las DLs a los sistemas de gestión de bases de datos, por ejemplo [Borgida95] [Baader *et al.* 02 (Cap 4 y 16)].

En cuanto a la relación de las DLs con otras lógicas, la observación inicial es que las DLs son subconjuntos de la lógica de primer orden, hecho conocido desde los orígenes de las DLs y estudiado extensamente en [Borgida96]. De hecho la DL $\mathcal{ALC}$ se corresponde con el fragmento de la lógica de primer orden restringiendo la sintaxis a las fórmulas que contienen dos variables. Este tipo de estudios han sido útiles para caracterizar y comparar la expresividad de distintas DLs.

5.4.2 DLs como formalismo de representación de ontologías

Existen numerosas ventajas derivadas del uso de las DLs como formalismo de representación de conocimiento, en general, y de conocimiento ontológico en particular, principalmente asociadas con sus mecanismos de razonamiento.

Por ejemplo, los mecanismos de clasificación automática de conceptos y el reconocimiento de instancias ayudan en el proceso de organización de la información, en la inferencia de conocimiento adicional, y facilitan la incorporación de nuevo conocimiento y el borrado de las definiciones innecesarias (con chequeo de incoherencias automático).

Estos mecanismos de razonamiento ayudan en el diseño de ontologías, en particular en el chequeo de consistencia en ontologías diseñadas por varios autores, y en su uso, en particular para determinar la consistencia de un conjunto de hechos respecto a la ontología y para determinar las instancias de las clases de la ontología.

Además, los mecanismos de clasificación también pueden ser utilizados en los procesos de acceso a la información y la capacidad de realizar consultas que corresponden a descripciones incompletas permiten realizar recuperaciones aproximadas. En sistemas de recuperación de información en los que cada objeto tiene una descripción compleja, el sistema puede recibir una consulta que describe objetos con una cierta estructura (encontrar comidas de dos platos, de forma que el primero .....). Las descripciones de los objetos que están representa-

dos en el sistema pueden clasificarse con respecto al resto de forma que los objetos similares se agrupan juntos. Además, son más similares cuanto más específico sea el nivel en el que están agrupados. Este mecanismo –utilizado por ejemplo en el sistema LASSIE [Devanbu *et al.* 91]– nos ofrece un esquema de indexación mucho más sofisticado que los esquemas basados en la aparición de ciertos valores.

Por otro lado, la detección de incoherencias ayuda en el proceso de adquisición del conocimiento inicial y en posteriores incorporaciones permitiendo verificar la consistencia del conocimiento representado. Esto facilita también la integración de ontologías, tanto para asertar relaciones entre distintas ontologías como para computar la integridad del resultado.

Por último, el hecho de que la sintaxis de las DLs sea muy cercana al lenguaje natural, junto con la definición precisa –mediante un lenguaje formal– de los términos usados en la representación –conceptos y relaciones– y su organización en una jerarquía de abstracción ayudan al usuario en la comprensión de la información. De este modo, un usuario que no esté familiarizado con el dominio puede obtener una visión comprensiva de los elementos del dominio y sus interrelaciones, tal y como han sido identificados por el experto que ha construido el sistema.

5.4.3 Tendencias actuales

Respecto a las tendencias actuales, en los últimos años la investigación en DLs ha estado guiada por el objetivo natural de su aplicación en distintas áreas: planificación, representación de acciones, Ingeniería del Software, bases de datos, integración de información, búsquedas inteligentes, Web Semántica, y sistemas CBR, entre otros. En [Baader *et al.* 02] se puede encontrar una descripción detallada de aplicaciones reales basadas en sistemas de DLs. La aplicación de sistemas de DLs en estas y otras áreas ha derivado en la necesidad de utilizar lenguajes expresivos. En concreto, la expresividad de la DL necesaria para razonar con modelos de datos y datos semi-estructurados ha contribuido a la identificación de numerosas extensiones para la aplicación práctica de las DLs.

Por ejemplo, los sistemas de integración de conocimiento requieren la posibilidad de representar las relaciones de inclusión entre relaciones y no sólo entre conceptos [Ullman97] [Calvanese *et al.* 98b]. Como veremos en el Capítulo 4, nuestra aproximación a la integración de ontologías se basa precisamente en esta capacidad que sí se incluye en LOOM.

En [Borgida95] se hace un estudio detallado de las ventajas de las DLs con respecto a la tecnología de gestión de información más ampliamente utilizada, los sistemas de gestión de bases de datos. El objetivo de capturar la semántica de los modelos de bases de datos para razonar con los esquemas conceptuales de datos ha marcado la importancia de las restricciones numéricas y los asertos con ciclos en las bases de conocimiento [Calvanese *et al.* 98a].

Los mismos requisitos también han surgido con la necesidad de representar ontologías en el contexto de la Web Semántica [Horrocks&Patel-Schneider01] [Horrocks02]. Los sistemas de DLs juegan un papel fundamental en la Web Semántica especialmente tras la unión DAML+OIL[24] [Horrocks02] en un lenguaje Web para representar ontologías basado en una lógica descriptiva (llamada $\mathcal{SHIQ}$), ampliada con enumeraciones (operador oneOf), tipos de datos *concretos* adicionales y con sintaxis basada en RDFS (RDF Squema). El editor OILEd ofrece una interfaz amigable basada en marcos y proporciona razonamiento a través del sistema de lógica descriptiva FaCT [Horrocks98].

[24] http://www.daml.org/2001/03/daml+oil-index

En las aplicaciones de las DLs más básicas se pone de manifiesto una de sus principales limitaciones: no integrar conocimiento (y en consecuencia tampoco razonamientos) sobre dominios específicos como números o cadenas de caracteres, que son necesarios en la mayoría de las aplicaciones. Por ejemplo, para modelar el concepto de "*personas jóvenes (entre 1 y 25 años)*" utilizaríamos un atributo simple edad y un valor concreto numérico (en un rango) que lo rellene. Para poder establecer relaciones de subsunción, por ejemplo, entre el concepto "*personas escolares (4-16)*" y "*personas jóvenes (<25)*", deberíamos utilizar propiedades de los intervalos numéricos. Los dominios concretos no solo incluyen tipos de datos simples, como números, sino otros dominios más elaborados, como regiones espaciales o intervalos temporales. Muchos trabajos han estudiado este tipo de extensiones de las DLs. Destacamos [Baader&Sattler98] donde se estudia el impacto de extender distintas DLs con dominios numéricos y funciones de agregación sobre ellos. El estudio está guiado por la necesidad de esta extensión en aplicaciones de DLs a los sistemas de bases de datos.

Relacionado con UML [Rumbaugh *et al.* 98], que es hoy en día el lenguaje estándar para llevar a cabo la fase de análisis del desarrollo del software y de los sistemas de información, una línea de trabajo que resultaría de gran utilidad sería el desarrollo de herramientas case que llevaran a cabo procesos de razonamiento automáticos sobre los esquemas UML, por ejemplo, para comprobar la consistencia o la redundancia de los mismos. Para capturar esquemas UML en DLs se requiere el uso de roles inversos, restricciones numéricas y puntos fijos generales en los conceptos para modelar las estructuras recursivas [Calvanese *et al.* 99].

Los beneficios que conlleva el uso de DLs en la representación de conocimiento han hecho que distintas comunidades, y en particular la comunidad de CBR, haya fijado su atención en ellas como tecnología de representación del conocimiento necesario en sus sistemas [Koelher94][Kamp96] [Napoli *et al.* 96] [Kamp97] [Lieber&Napoli98][Salotti&Ventos98]. Por la relación con nuestro trabajo describimos estas aproximaciones en el apartado siguiente, que repasa las ventajas derivadas del uso de las DLs para los sistemas KI-CBR.

En la evolución de los sistemas de DLs se observa que una consolidación del espíritu de aumentar la expresividad de sus lenguajes (que es la aproximación de LOOM) ya que los requisitos surgidos de la aplicación de las DLs en los distintos ámbitos han estimulado la necesidad de incorporar mecanismos de representación más expresivos y adaptar o extender consecuentemente las técnicas de razonamiento asociadas. Según la opinión expresada en [Calvanese *et al.* 01] la investigación en DLs se está centrando en lenguajes terminológicos mucho más expresivos, donde la propiedad que se desea mantener no es la tratabilidad del razonamiento sino la decibilidad.

5.4.4 Uso de las DLs en los sistemas KI-CBR

Entre los trabajos de la comunidad de CBR que han fijado su atención en las DLs como tecnología de representación del conocimiento podemos distinguir dos líneas de investigación. Una línea más teórica en la que enmarcamos los trabajos de G. Kamp [Kamp96] [Kamp97] se refiere al estudio de la expresividad de la DLs en el contexto de los sistemas CBR. Realmente se centra en el estudio de los dominios concretos y su inclusión en los sistemas de DLs. En [Kamp96] extiende una DL con un dominio concreto que permite manejar sistemas de inecuaciones lineales. En [Kamp97] se aplica el marco general presentado en [Kamp96] al dominio de recuperación de datos bibliográficos. Estudian qué dominios concretos son adecuados para manejar preguntas que integran datos puramente bibliográficos con información del contenido de un artículo. Presentan una jerarquía de dominios numéricos admisibles así como diversos dominios concretos sobre cadenas de caracteres y textos y cómo se pueden reducir relaciones *espaciales* a dominios concretos numéricos.

También los trabajos presentados en [Ventos *et al.* 98] [Coupey *et al.* 98] se pueden enmarcar en esta línea. Su principal objetivo es definir y *clasificar* conceptos que incluyan conocimiento por defecto y excepciones y extender el mecanismo de clasificación de forma adecuada, para después proponer procesos de recuperación y selección de casos basados en esa operación de clasificación. En [Ventos *et al.* 98] se describe C-CLASSIC$_{\delta\varepsilon}$ una extensión de C-CLASSIC en la que se añaden las conectivas δ (por defecto) y ε (excepción) para incluir el comportamiento por defecto y sus excepciones en las definiciones de los conceptos, manteniendo la clasificación de conceptos monótona y polinomial. En [Coupey *et al.* 98] la lógica descriptiva utilizada es $\mathcal{ALN}_{\delta\varepsilon}$,. Describen su sintaxis, su semántica formal y los procesos del CBR que se basan en el *algoritmo de subsunción (*para $\mathcal{ALN}_{\delta\varepsilon}$) que permite la clasificación de conceptos que incluyan en sus descripciones comportamiento por defecto y excepciones.

La otra línea de trabajo [Koelher94] [Koelher96] [Napoli *et al.* 96] [Napoli *et al.* 97] [Salotti&Ventos98], en la que también se enmarca nuestra propuesta, no se centra en el estudio teórico de las DLs, ni de las condiciones o extensiones de las DLs que le permiten a una DL ser adecuada para CBR. En vez de eso, se parte de los mecanismos de razonamiento de un sistema de DL, más o menos expresivo, y se estudia cómo formular los procesos CBR en base a los mecanismos de razonamiento básicos de cualquier sistema de DLs.

Uno de los primeros trabajos que aplicaron las DLs a los sistemas CBR, fueron los de J. Koelher [Koelher94] [Koelher96] donde se construye un sistema de planificación CBR. Los casos representan planes para realizar operaciones en el sistema de correo electrónico de Unix, que se indexan a través de las precondiciones y postcondiciones que se representan mediante conceptos de un sistema de DLs. La representación de casos no se basa únicamente en DLs sino que para la solución, el conjunto de pasos del plan, utiliza fórmulas en un sistema de lógica temporal con operadores modales. Define dos tipos de recuperación. La recuperación fuerte busca casos cuyo índice *implica* el del problema consulta, lo que significa, que las precondiciones de la consulta implican las del caso y las postcondiciones del caso implican las del problema. Si esto no es posible se aplica un proceso de recuperación débil que busca los casos cuyas precondiciones sean implicadas por las precondiciones de la consulta y no establece condición para las postcondiciones. Las precondiciones y postcondiciones, tanto de la consulta como de los casos, se representan como conceptos de DLs y las implicaciones corresponden a la relación de subsunción entre los conceptos: A implica B significa que A es subsumido por B.

Los trabajos de A. Napoli [Napoli96] [Napoli *et al.* 97] toman como punto de partida el trabajo de J. Koelher aplicándolo en primer lugar a los sistemas de marcos y posteriormente a DLs. También consideran el proceso de recuperación como un proceso de clasificación del índice de la consulta (un concepto que representa sus precondiciones y objetivos) en una jerarquía pero permitiendo localizaciones más flexibles que recorren ascendente o descendentemente la jerarquía de precondiciones y objetivos.

El trabajo presentado en [Salotti&Ventos98] también utiliza los mecanismos de razonamiento de un sistema de DLs (C-CLASSIC) para definir un proceso de recuperación de casos. Proponen la representación de casos como individuos que están indexados a través de conceptos organizados en una taxonomía y que han sido construidos de forma manual por el diseñador del sistema CBR. Para recuperar casos similares a una consulta, proponen la construcción de un individuo que represente las características de la consulta y su reconocimiento para obtener los índices (conceptos) de los que es instancia. Los casos que comparten al menos un índice con la consulta, es decir, son instancia de al menos uno de los conceptos de los que es instancia la consulta son recuperados como casos relevantes. El inconveniente de

este tipo de valoración de la similitud es que en algunas ocasiones podría resultar demasiado relajada, recuperándose un número muy elevado de casos. Otra opción más estricta podría consistir, por ejemplo, en considerar los casos que compartan todos los índices con la consulta. Otra opción es utilizar un proceso de selección de casos que elija del conjunto de casos localizados aquellos que sean más similares. Para ello proponen una aproximación -que hemos descrito en el Capítulo 2 (Apartado 4.2.3.1)- que se basa en computar el LCS entre la consulta y cada uno de los casos localizados anteriormente.

Además de las ventajas de las DLs para la representación del conocimiento terminológico de un sistema KI-CBR, desde el punto de vista de los procesos CBR son destacables las siguientes características:

- Su capacidad para construir descripciones estructuradas de los casos, proporcionando una forma expresiva de representarlos.
- Aplicadas a la representación de los índices, permiten definir éstos con una semántica formal que permite organizarlos en una jerarquía de abstracción que se construye a partir de sus definiciones y que facilita la comprensión de los atributos utilizados en la indexación de los casos.
- Los mecanismos de clasificación de conceptos y reconocimiento de instancias, junto con la capacidad de detectar incoherencias, facilitan la construcción y extensión automáticas del esquema de indexación.
- Esos mecanismos pueden ser utilizados para la recuperación de casos
  - Se puede construir una descripción de un concepto a partir de las restricciones establecidas en la consulta, dicho concepto se clasifica en la taxonomía de conceptos, y se obtienen los individuos del sistema que son instancias suyas.
  - Se puede construir un individuo genérico sobre el que se realizan asertos correspondientes a las restricciones especificadas en la consulta. Gracias al razonamiento hacia adelante, se pueden inferir hechos adicionales que enriquecen la consulta original (compleción de instancias). Seguidamente se reconoce el individuo y se recuperan los restantes individuos que son instancias del concepto más específico del que el individuo consulta es instancia.

6. Resumen y conclusiones del capítulo

En este capítulo hemos descrito las ontologías y los PSMs, dos tipos de componentes de conocimiento presentes en las principales metodologías de desarrollo de KBSs. Estas metodologías plantean el desarrollo de los KBSs como un proceso de reutilización de componentes de conocimiento, tanto conocimiento del dominio como de resolución de problemas, así como la integración y configuración de dichos componentes.

Una de las propuestas de esta tesis consiste en promover la reutilización de conocimiento proporcionando y utilizando bibliotecas de componentes para el diseño de aplicaciones KI-CBR, en particular ontologías y PSMs. En este capítulo hemos presentado algunos conceptos generales para introducir dichos componentes, haciendo hincapié sobre los aspectos relativos a las ventajas y dificultades de su reutilización en aplicaciones.

Como formalismo de representación de conocimiento para formalizar las ontologías y los PSMs hemos elegido las DLs que, además de capacidades de representación adecuadas, ofrecen mecanismos de razonamiento con el conocimiento representado. En este capítulo hemos descrito las características principales de las DLs y hemos justificado los aspectos que hacen

de las DLs un formalismo idóneo para la representación de conocimiento ontológico y para el desarrollo de sistemas KI-CBR como un tipo especial de KBSs.

En los capítulos siguientes describimos nuestra propuesta utilizando las ideas descritas de forma general en este capítulo, en cuanto a la construcción y reutilización de ontologías y PSMs. En concreto, hemos utilizado LOOM, un sistema de DLs, para formalizar CBROnto, una ontología con conocimiento sobre CBR que incluye una biblioteca de PSMs reutilizables que aprovechan los mecanismos de razonamiento de las DLs. En el Capítulo 6, describimos la arquitectura del sistema COLIBRI que propone una metodología para diseñar aplicaciones KI-CBR basada en la reutilización de terminología sobre el dominio de una biblioteca de ontologías y en la reutilización de terminología sobre CBR de CBROnto.

Capítulo 4
CBROnto: Una ontología para CBR

1. Introducción

En el capítulo anterior hemos presentado el interés de compartir y reutilizar conocimiento, y dos aproximaciones que resultan útiles para ello: las ontologías y los métodos de resolución de problemas (PSMs). Una idea subyacente a esta tesis ha sido la de aplicar estos dos tipos de componentes al diseño de sistemas KI-CBR: la reutilización de ontologías como fuente de conocimiento sobre el dominio de aplicación y sobre CBR, y los PSMs para representar el conocimiento independiente del dominio asociado con los procesos CBR que se reutilizan en los sistemas CBR diseñados. En esta línea, la aportación fundamental de nuestro trabajo es la representación explícita, de forma declarativa, de conocimiento ontológico sobre la resolución de problemas mediante CBR, tanto respecto al vocabulario y terminología como a las tareas y métodos típicamente asociados a los sistemas CBR.

Durante el diseño de la arquitectura de COLIBRI surgió la necesidad de disponer de terminología unificadora sobre CBR. Esto nos llevó a definir CBROnto, una ontología que captura términos semánticamente importantes para los sistemas CBR. Las definiciones de CBROnto son relativas a CBR en general, y son independientes del dominio de aplicación. El diseño de una nueva aplicación CBR se basa, por un lado, en reutilizar terminología sobre el dominio de una biblioteca de ontologías y, por otro lado, en reutilizar terminología sobre CBR de CBROnto. Además de la terminología, CBROnto incluye conocimiento sobre las tareas y los métodos CBR, es decir, incluye representaciones explícitas de ciertos métodos que resuelven las tareas involucradas en el CBR y que pueden ser reutilizados en distintos dominios. Los métodos de CBROnto están definidos en base a la terminología CBR sin referirse a un dominio concreto, es decir, son específicos de las tareas CBR pero independientes del dominio de aplicación.

Para que estos métodos genéricos puedan ser reutilizados en una aplicación y dominio concretos será necesario relacionar la terminología CBR usada en la definición de los métodos con la terminología del dominio. En este capítulo describiremos cómo la terminología de CBROnto nos sirve como esquema integrador entre los métodos y el conocimiento del dominio que hemos adquirido a partir de ontologías y que no fue específicamente diseñado para ser usado en una aplicación CBR. El mecanismo que utilizamos para llevar a cabo esta

integración de conocimiento es la clasificación semántica de las DLs. Suponiendo que disponemos de un modelo terminológico del dominio, CBROnto proporciona la infraestructura que sirve como "pegamento" sintáctico y semántico entre esta terminología del dominio y los métodos CBR genéricos que se reutilizan.

En este capítulo describimos las ideas básicas de CBROnto y algunos aspectos de su implementación, aunque los detalles de su formalización en LOOM se incluyen en el Apéndice B. Tras la motivación inicial que aparece en el Apartado 2, el Apartado 3 describe el conocimiento terminológico sobre las tareas y métodos CBR de CBROnto que se particulariza en una biblioteca de PSMs (que se detalla en el Capítulo 5 de esta memoria). El Apartado 4 describe CBROnto como una ontología que incluye, por un lado, terminología sobre CBR -haciendo hincapié en el lenguaje de descripción de casos- y por otro lado, una biblioteca de PSMs definidos con ese vocabulario. El Apartado 5 trata la organización de la base de casos en una estructura que facilite el acceso a los mismos. Para esta tarea hemos estudiado el uso de una técnica inductiva que se aplica a la base de casos: el Análisis Formal de Conceptos.

2. Utilidad de una ontología para CBR

Antes de describir los detalles de CBROnto queremos motivar su construcción con un ejemplo. Supongamos que en un sistema CBR para una agencia de viajes cada caso representa un viaje de su catálogo anual, cuánta gente lo ha llevado a cabo y su opinión sobre él. El análisis de la aplicación lleva al diseñador a proponer una estructura de casos con descripción pero que no tienen una solución asociada y que incluye, como resultado del viaje, el número de personas que lo han realizado satisfactoriamente, los que han tenido alguna queja y los que no han opinado. Aunque hay distintas posibilidades podríamos pensar en una representación como la siguiente usando LOOM:

```
(defconcept CASO-VIAJE :is-primitive
                (:and   CASO
                        (:the descripcion VIAJES)
                        (:at-most 0 solucion)
                        (:the resultado OPINION)))
(defconcept VIAJES :is-primitive
                (:and   (:at-least 1 tipo)(:all tipo ACTIVIDADES)
                        (:at-least 1 destino)(:all destino DESTINOS)
                        (:all estacion ESTACIONES)
                        (:all transporte TRANSPORTES)
                        (:at-least 1 duracion)
                        (:all duración (:and Number (:through 1 365))
                        (:the precio (:and Number (:through 30 1000000)))))
(defconcept ACTIVIDADES)
(tell (:about aventuras ACTIVIDADES))  (tell (:about cultural ACTIVIDADES))
(tell (:about esqui ACTIVIDADES))      (tell (:about descanso ACTIVIDADES))
(tell (:about educacion ACTIVIDADES))
(defconcept ESTACIONES)
(tell (:about primavera ESTACIONES)) (tell (:about verano ESTACIONES))
(tell (:about otoño ESTACIONES)) (tell (:about invierno ESTACIONES))
(defconcept DESTINOS) …
(defconcept TRANSPORTES) …
(defconcept OPINION :is-primitive
        (:and   (:the positiva (:and Number (:through 0 100))
                (:the negativa (:and Number (:through 0 100))
                (:the no_contesta (:and Number (:through 0 100))))
```

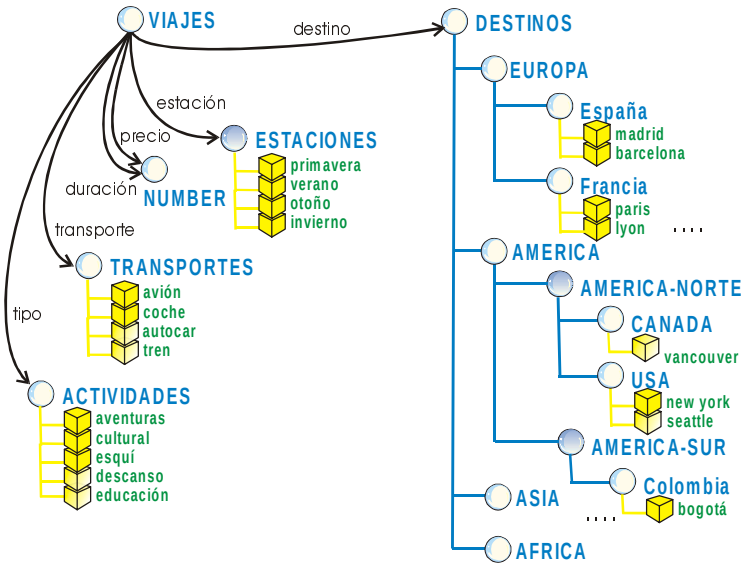


Figura 4-1. *Conocimiento terminológico sobre el dominio Viajes.*

```
------------------- Descripción de un caso concreto -----------------------
(tell   (:about caso1 CASO-VIAJE
                      (descripcion caso1 viaje1) (resultado caso1 opinion1))
(tell   (:about viaje1 VIAJE (tipo educacion)(transporte avion)(destino Madrid)
                             (estacion Verano)(duracion 20)(precio 1000))
(tellm (:about opinion1 OPINIÓN)(positiva 60)(negativa 20)(no_contesta 20))
```

En el ejemplo se observa que un mismo sistema de representación de conocimiento, en este caso un sistema de DLs como LOOM, permite de forma integrada describir una ontología (o base de conocimiento) simple sobre el dominio de los viajes (Figura 4-1) y describir casos que utilizan dicha ontología. Además, se pone de manifiesto la existencia de un cierto lenguaje de descripción de casos en el que identificamos:

- Componentes del lenguaje LOOM como `defconcept`, `tell`, `:and`, `:all`, `:some`, `:at-least`, entre otros. Se utilizan como constructores del formalismo de representación elegido, que es vacío de contenido en sí mismo y necesita conocimiento de algún dominio para poder razonar.
- Componentes del dominio de los viajes como `DESTINOS`, `ACTIVIDADES`, `TRANSPORTES`, `ESTACIONES`, `duración`, `estación`, `verano`, `otoño`, `primavera`, `verano`, `avión`, `coche`, `precio`, etc. Estos elementos representan el tipo de conocimiento que esperaríamos encontrar en una ontología sobre viajes o vacaciones.
- Componentes de esta aplicación en concreto como `opinion`, `positiva`, `negativa`, o `no_contesta`. Al contrario que los elementos del dominio de los viajes, estos elementos no esperaríamos encontrarlos en una ontología sobre viajes porque son demasiado dependientes del problema concreto que nos ocupa.
- Componentes relacionados con el CBR como `descripcion`, `solucion`, `resultado` o `caso`, que permiten identificar qué individuos del dominio representan los casos, qué parte suya corresponde a la descripción sobre la que computar la similitud, etc.

En el ejemplo, las instancias del concepto CASO-VIAJE son los casos, y las instancias del concepto VIAJES representan las descripciones de los casos.

La primera motivación para desarrollar una ontología sobre CBR surge de este último tipo de componentes asociados al lenguaje de descripción de casos que, implícitamente, hemos utilizado en este ejemplo. El objetivo subyacente es la definición de un lenguaje para describir los elementos que intervienen en un sistema CBR y utilizar esta terminología para definir métodos genéricos de resolución de problemas.

Nuestra propuesta construye métodos CBR independientes del dominio que hacen referencia a la terminología de la ontología CBR. Por ejemplo, un posible método para recuperar el *caso* más *similar* a una *consulta*, consiste en calcular la *similitud* entre la *consulta* y la *descripción* de cada *caso* almacenado y devolver aquel *caso* para el que se obtenga un *valor de similitud* mayor. Las palabras marcadas en cursiva representan terminología sobre CBR que no está asociada con ninguna aplicación o dominio concretos.

Esta terminología representa el vocabulario sobre CBR al que los métodos de CBROnto hacen referencia. Para que un método pueda aplicarse en un dominio habrá que concretar qué es un caso, qué es una consulta, qué medida de similitud se usa o qué parte del caso corresponde a su descripción, lo que requiere un esfuerzo de integración adicional. En nuestro ejemplo de la agencia de viajes, una *consulta* se planteará como una instancia del concepto *VIAJES* dando todos o alguno de sus descriptores (tipo, duración, destino, estación y transporte). Usando una cierta *medida de similitud* se devuelve aquella instancia de *CASO-VIAJE* cuyo *VIAJE* asociado sea más similar al planteado en la *consulta*.

2.1 CBROnto como núcleo de COLIBRI

En la Figura 4-2 se muestra el esquema de una aplicación diseñada usando COLIBRI, donde CBROnto es el núcleo en torno al cual se estructura el conocimiento de la aplicación [Díaz&González00a] [Díaz&González01e] [Díaz&González01f] [Díaz&González01g]. Aunque en el Capítulo 6 se detalla el proceso de diseño de aplicaciones usando COLIBRI, en este apartado realizamos una breve introducción para enmarcar las distintas facetas de CBROnto que se describen en este capítulo.

En nuestra arquitectura clasificamos el conocimiento en tres tipos: conocimiento terminológico sobre el dominio, conocimiento de experiencias previas (los casos) y conocimiento sobre CBR incluyendo terminología y métodos.

Como primer paso del proceso de diseño de una aplicación CBR se construye el modelo de conocimiento del dominio reutilizando, cuando sea posible, ontologías previamente formalizadas o modelando el dominio desde cero si no se tiene acceso a ontologías adecuadas. El conocimiento ontológico del dominio es la terminología –el lenguaje– que utilizaremos para representar cualquier otro tipo de conocimiento en ese dominio. En concreto, este vocabulario, complementado con el lenguaje de definición de casos de CBROnto, se utilizará para la definición de los casos. El uso de ontologías del dominio facilita este proceso y permite la construcción de bases de casos que comparten cierto vocabulario estandarizado.

El siguiente paso es la reutilización y configuración de los métodos CBR, para lo cual será necesario llevar a cabo un proceso de integración de los términos del dominio en torno a los términos de CBROnto. Este proceso de integración permite que los métodos que están descritos de forma independiente del dominio –porque hacen referencia únicamente a los términos de CBROnto– hagan un uso efectivo del conocimiento del dominio. Proponemos una arquitectura de dos capas donde en la capa inferior hay conocimiento dependiente del domi-

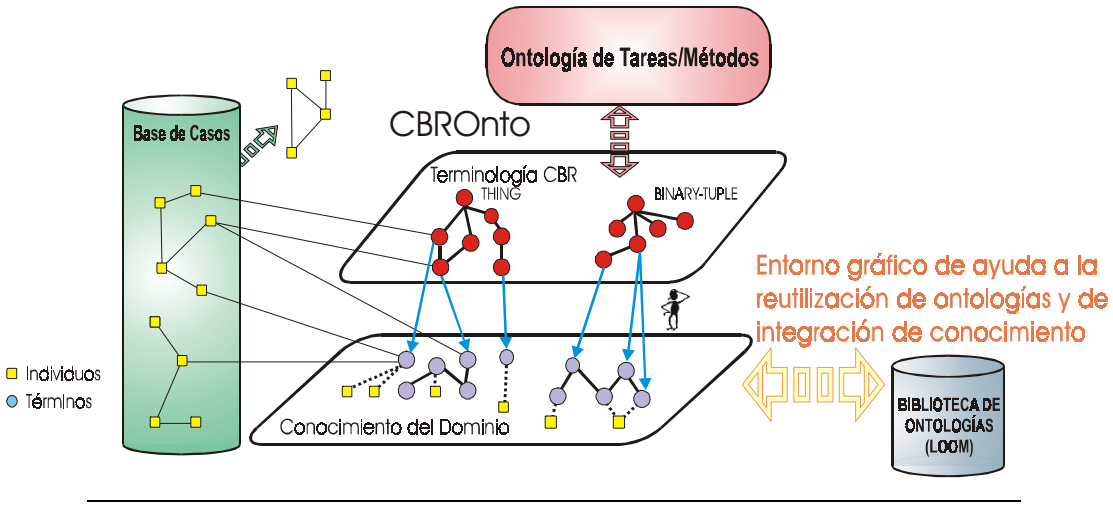


Figura 4.2. *Arquitectura de una aplicación CBR diseñada con COLIBRI*

nio, y la terminología de la capa superior se usa como puente de conexión entre el conocimiento del dominio y los PSMs genéricos. De esta forma, la teoría específica del dominio es intercambiable y el mismo conocimiento puede jugar distintos papeles en distintos contextos de resolución de problemas.

El proceso de integración entre la terminología del dominio y la terminología utilizada por los métodos CBR se basa en la clasificación de términos de las DLs. En concreto, la integración se basa en la clasificación de los términos del dominio bajo los términos de CBROnto utilizados por los métodos. Se puede observar que nuestra aproximación comparte varias ideas con las principales metodologías de soporte para el diseño de KBSs descritas en el Capítulo 3: la reutilización de componentes de conocimiento y la división del conocimiento de un KBS en varios modelos según el papel que juegan, en concreto la desvinculación del conocimiento del dominio y de los métodos que lo utilizan.

Nuestra aproximación también se relaciona con UPML [Fensel *et al.* 98a], una arquitectura software para la reutilización de KBSs que hemos descrito en el Capítulo 3 y que se basa en la reutilización de PSMs, y con la arquitectura ABC (*Adaptors-Bridges-Connectors*) [Plaza&Arcos00] que propone un esquema de descripción utilizando las ideas de UPML aplicadas a los sistemas CBR. En concreto ABC usa tres elementos tomados de UPML: la descripción de la tarea que va a resolver el sistema CBR, el modelo del dominio y una biblioteca de adaptadores, donde los adaptadores juegan el papel que los PSMs juegan en UPML.

La arquitectura de COLIBRI comparte ideas con UPML, en concreto, comparte el proceso de desarrollo de KBSs como configuración de componentes reutilizables y su formulación en términos de tareas, métodos y un modelo de conocimiento específico del dominio. Aunque esta aproximación también la comparte la arquitectura ABC, existe una diferencia fundamental en el planteamiento de la resolución de problemas en ABC y el de COLIBRI. En ABC se sigue la vista de resolución de problemas como construcción de modelos, es decir, para resolver un problema se construye un modelo especifico del problema que satisface los requisitos de corrección establecidos por la tarea que queremos resolver. Con este punto de vista, un KBS recibe un modelo de entrada (parcial) y se usan los adaptadores (usando el conocimiento del dominio y los casos) para ampliar este modelo hasta que se construya un modelo correcto y completo respecto a la especificación de los objetivos de la tarea.

3. Tareas y métodos de CBROnto

En el Capítulo 3 se definieron los PSMs como "componentes reutilizables que describen algún proceso de razonamiento de forma independiente del dominio y de la implementación". En este apartado estudiaremos la faceta de CBROnto que corresponde a la representación explícita de métodos asociados a las tareas CBR que puedan ser reutilizados en distintos dominios [Díaz&González02]. El objetivo es poder expresar la parte operacional de COLIBRI en términos de las tareas y los métodos que representan los procesos que típicamente se llevan a cabo en un sistema CBR.

El Apartado 3.1 repasa las tareas involucradas en el CBR y algunas estructuras de tareas propuestas en la literatura. En el Apartado 3.2 introducimos los elementos terminológicos que se utilizan para representar los PSMs y sus necesidades de conocimiento, y la terminología necesaria para organizarlos por tareas. El método CBR -que es un PSM en sí mismo- se describe en el Apartado 3.3 junto con una descripción de alto nivel de los métodos que resuelven cada una de las subtareas derivadas de su aplicación: recuperación, adaptación, revisión y aprendizaje.

3.1 Ontología de tareas CBR

En [Aamodt&Plaza94] se presenta un marco analítico general descriptivo de los principios metodológicos y fundamentos del CBR, que está influenciado por las metodologías para describir los sistemas inteligentes al nivel de conocimiento, en particular por la metodología *Components of Expertise* [Steels90]. En este apartado describimos este y otros marcos relacionados que han influido en la definición de CBROnto.

En el Capítulo 2 hemos descrito el CBR como un paradigma de resolución de problemas que utiliza el conocimiento específico de experiencias previas de resolución de problemas: los casos. Un nuevo problema se soluciona buscando un caso previo similar y reutilizándolo en la nueva situación. Además, cada nueva experiencia de resolución se almacena y estará disponible para la resolución de problemas futuros.

El CBR en sí mismo se puede considerar un método para resolver problemas –un PSM– que descompone la tarea principal en varias subtareas que serán resueltas a su vez por otros métodos. Por ejemplo, recuperar casos adecuados, adaptar casos, aprender casos, integrar un nuevo caso en la estructura de conocimiento, organizar la estructura de casos de manera que sea sencillo encontrar un caso adecuado en un episodio posterior de resolución de un problema, etc. Es importante incidir en la idea de que "resolución de un problema" se entiende en un sentido amplio en el que no significa únicamente encontrar una solución concreta a un problema, sino que puede referirse a actividades como justificar o criticar una solución propuesta por el usuario o interpretar una situación.

Los siguientes apartados describen algunas de las estructuras de tareas encontradas en la literatura así como las características de la estructura de tareas de CBROnto, respectivamente.

3.1.1 Las tareas del CBR

Las tareas a resolver cuando se usa el paradigma de resolución de problemas basado en casos son básicamente: identificar las características relevantes de la situación o problema actual, encontrar un caso anterior similar al actual, usar el caso recuperado para sugerir una solución al problema actual, evaluar la solución propuesta y corregirla si es necesario, y actualizar el sistema aprendiendo de esta experiencia.

Aunque estas tareas se consideran más o menos comunes a todos los sistemas CBR, existen muchos aspectos que varían considerablemente entre las distintas aproximaciones al CBR, por ejemplo, ¿cómo llevar a cabo estas tareas? ¿cuál es la fase del proceso de resolución en la que se hace más hincapié? ¿qué tipo de problemas se pretende resolver? Lo que se entiende por paradigma CBR agrupa distintas formas de resolver las tareas involucradas. En concreto, distintos métodos para organizar, recuperar, utilizar e indexar el conocimiento almacenado de los casos resueltos.

En [Aamodt&Plaza94] se lleva a cabo una clasificación de estos métodos en tipos. Algunos métodos utilizan casos que representan experiencias concretas y otros usan casos generalizados a partir de otros casos similares; algunos métodos indexan los casos con un vocabulario prefijado y otros con un vocabulario abierto; otros métodos pueden no indexar los casos; y otros métodos pueden indexar los casos en una estructura plana o jerárquica. Además, algunos métodos aplican la solución de un caso previo directamente y otros métodos la modifican según las diferencias entre el caso actual y el caso recuperado. Algunos métodos usan un modelo complejo de conocimiento general del dominio para guiar las tareas de recuperación de casos, adaptación de las soluciones y aprendizaje de la experiencia. Otros métodos pueden utilizar conocimiento más básico, compilado o incluso ningún conocimiento adicional, aparte del que representan los propios casos. Por último, algunos métodos usan conjuntos de casos muy grandes y variados mientras que otros métodos se basan en un conjunto limitado de casos prototípicos.

Es evidente que los distintos tipos de métodos, todos ellos agrupados bajo el paradigma CBR, llevan a distintos tipos de sistemas aunque todos ellos comparten la misma filosofía. El marco propuesto en [Aamodt&Plaza94] para describir los métodos y sistemas CBR tiene dos partes: un *modelo de procesos* del ciclo CBR, que ha sido aceptado por la comunidad de CBR como modelo unificador y que hemos descrito en el Capítulo 2, y una *estructura de tareas-métodos* para CBR que es complementaria del modelo de procesos y que es objeto de nuestro interés en este apartado.

El modelo de procesos representa un modelo dinámico que identifica los subprocesos principales de un ciclo CBR, sus dependencias y sus resultados. Se utiliza para enfatizar la noción del CBR como un ciclo de etapas secuenciales. La estructura de tareas-métodos es una vista orientada a las tareas del CBR donde se describe una descomposición de tareas así como distintos tipos de métodos CBR que resuelven esas tareas. En esta estructura cada proceso del ciclo CBR: recuperar, reutilizar, revisar y almacenar, se considera una tarea que el sistema CBR tiene que resolver. Esto facilita la descripción de los métodos asociados a las tareas desde el punto de vista del razonador.

La idea subyacente es la de llevar a cabo un modelado a nivel de conocimiento [Newel82] considerando que un sistema es un agente que se plantea ciertos objetivos y dispone de ciertos medios para lograr esos objetivos. La descripción del sistema se puede hacer desde el punto de vista de las tareas a resolver, los métodos para hacerlo y el conocimiento del dominio del que dispone. Las tareas están determinadas por los objetivos del sistema y cada una de las tareas se resuelve aplicando uno o más métodos.

La Figura 4-3 presenta la estructura de tareas y métodos propuesta en [Aamodt&Plaza94]. Cada nodo representa una tarea o un método y los nombres de los métodos aparecen con efecto de cursiva. Las aristas que unen nodos de tareas representan descomposiciones de tareas en subtareas, es decir, relaciones `parte-de`. La tarea que ocupa el nodo raíz es la tarea principal de "resolver problemas y aprender de la experiencia" y el método para resolver esa tarea es el "razonamiento basado en casos". El método CBR indica que la tarea principal se divide en cuatro subtareas: recuperar (*retrieve*), adaptar (*reuse*), revisar (*revise*) y aprender (*retain*).

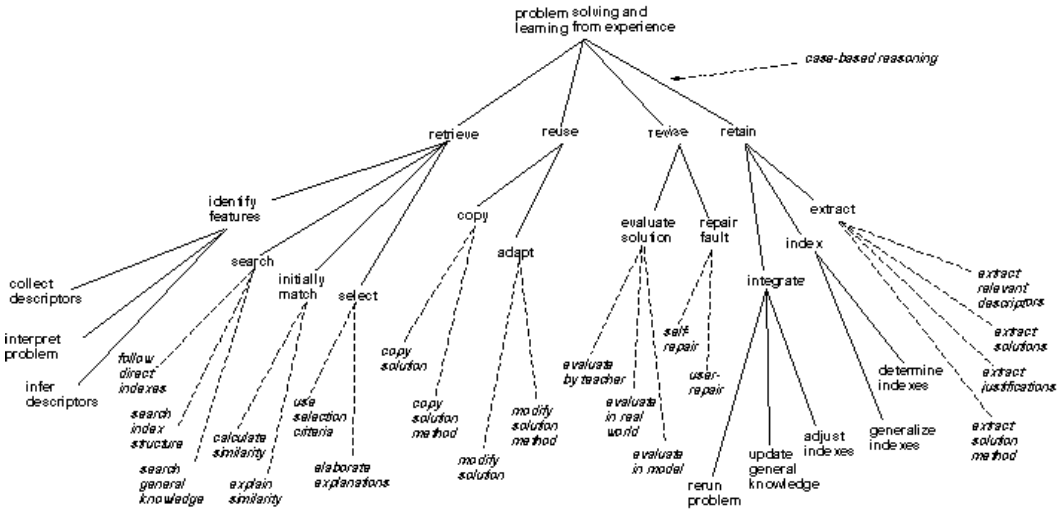


Figura 4-3. *Estructura de Tareas y Métodos [Aamodt&Plaza94]*

Cada subtarea se subdivide a su vez en otras subtareas según indique un cierto método. Por ejemplo, recuperar se divide a su vez, usando algún método de recuperación, en las subtareas: "identificación" de descriptores relevantes (*identify features*), "búsqueda" de un conjunto de casos resueltos similares (*search*), "encaje inicial" entre los descriptores del caso nuevo y los casos resueltos (*initially match*), y "selección" del caso más similar (*select*).

La figura no muestra ninguna estructura de control sobre las subtareas, aunque sí se indica un orden de secuencia lógica según el orden de escritura, ya que el control real se debe especificar como parte del método. La relación entre tareas y métodos, representada por las líneas discontinuas, es uno a muchos ya que se pueden identificar varios métodos alternativos para resolver una misma tarea. Cada método especifica el algoritmo que controla la ejecución de las subtareas y utiliza el conocimiento que necesite. Los métodos de la figura son clases de métodos de alto nivel de los que los sistemas CBR utilizan métodos concretos. La especificación de métodos en la figura no es exhaustiva ni completa, ya que algunas tareas se resuelven usando un solo método, para otras será necesario combinar varios métodos y para otras puede haber métodos alternativos. Los métodos que descomponen una tarea en subtareas son métodos de descomposición –que no aparecen explícitamente en la figura–, como el método CBR en sí mismo. En el nivel inferior de la jerarquía deberían aparecer los métodos de ejecución que resuelven directamente una tarea.

Nuestra aproximación se refiere precisamente a especificar y formalizar en nuestro sistema algunos de estos métodos –tanto de descomposición como de ejecución– que resuelven tareas CBR. Los métodos se representan de forma declarativa utilizando el lenguaje de descripción de métodos de CBROnto y su especificación operacional se implementa usando Lisp/LOOM. Una implementación ejecutable como la que proponemos es necesaria cuando el objetivo de reutilizar los métodos es construir sistemas que funcionen. Como ocurre en general en otros campos de la Inteligencia Artificial, no existen métodos CBR universales adecuados para todos los dominios de aplicación. Por tanto, una de las motivaciones principales de esta tesis ha sido la de plantear un sistema que ofrece distintas alternativas para re-

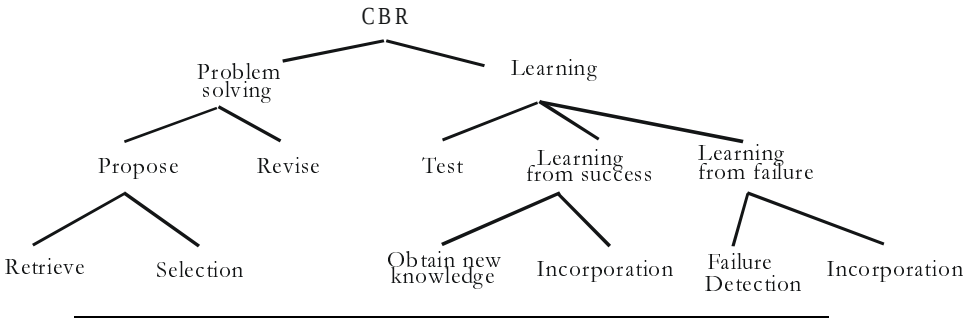


Figura 4-4. *Estructura de Tareas [Armengol&Plaza93]*

solver las tareas CBR en distintos dominios y aplicaciones, es decir, ofrece métodos variados que pueden aplicarse a distintas situaciones y facilitan el prototipado de sistemas CBR.

Los problemas abordados por la investigación de la comunidad de CBR se pueden agrupar en cinco áreas principales: representación de conocimiento, métodos de recuperación, métodos de reutilización, métodos de revisión de soluciones y métodos de aprendizaje. En la literatura existen muchos métodos que resuelven las tareas CBR, algunos de los cuales hemos descrito en el Capítulo 2. No es un objetivo de esta tesis llevar a cabo un análisis exhaustivo al respecto, sino presentar un marco en el que todos esos métodos podrían integrarse y estudiar aquellos métodos que saquen partido de los mecanismos de razonamiento de las DLs y del conocimiento terminológico que complementa el conocimiento de los casos.

Existen otros trabajos de análisis del CBR al nivel de conocimiento. Por ejemplo, el trabajo presentado en [Goel96] o el modelo de B. Fuchs sobre la tarea de adaptación [Fuchs99]. Más relacionado con el trabajo de A. Aamodt y E. Plaza en [Armengol&Plaza93] los autores presentan un estudio similar, aunque se centran principalmente en el análisis de los métodos de aprendizaje de distintos sistemas. Del análisis a nivel de conocimiento de tres sistemas CBR clásicos –CHEF, PROTOS y CASEY– obtienen una estructura de tareas común. Estos sistemas tienen en común la característica, relevante en el contexto de esta tesis, de utilizar modelos generales relativamente complejos de conocimiento del dominio. La Figura 4-4 muestra la estructura de tareas propuesta en [Armengol&Plaza93]. Los nodos representan tareas y las aristas descendentes significan descomposiciones de las tareas en subtareas.

Observamos que las estructuras de tareas CBR propuestas en los dos modelos anteriores son similares, aunque en [Armengol&Plaza93] se hace mas hincapié en el análisis del aprendizaje y menos en el resto de las tareas. El interés de su análisis, y el nuestro propio, es el de clarificar el papel de la adquisición, uso y aprendizaje de conocimiento general sobre el dominio en las distintas tareas CBR. En [Arcos&Plaza93] se presenta la arquitectura MMA (*Massive Memory Architecture*) como un marco computacional que da soporte a este tipo de análisis a nivel de conocimiento que extrae estructuras de tareas comunes para los sistemas CBR. Aunque MMA está relacionada con CBROnto, una diferencia fundamental es precisamente el uso de conocimiento del dominio, ya que ellos suponen un modelo de analogías en el que, aparte de los casos, no existe conocimiento adicional sobre el dominio de aplicación, y nuestros métodos se definen por el uso intensivo de conocimiento sobre el dominio.

3.1.2 La estructura de tareas de CBROnto

El núcleo de nuestra propuesta no se centra en el modelo de tareas sino en los métodos, por tanto, nuestra estructura de tareas no es novedosa sino que es similar a la propuesta en [Aa-

modt&Plaza94] sobre la que hay acuerdo en la comunidad CBR. Nuestro interés se centra en estudiar y formalizar algunos métodos alternativos que resuelven las tareas básicas del CBR. Nos hemos centrado en aquellos métodos CBR que, además del conocimiento de los casos, usan de forma intensiva el conocimiento general sobre el dominio disponible en el sistema CBR y sacan partido de los mecanismos de razonamiento de las DLs.

Una diferencia fundamental de nuestro trabajo con los anteriores, es que las estructuras que tradicionalmente se han propuesto se centran fundamentalmente en las tareas y no en los métodos para resolverlas. Los modelos de [Armengol&Plaza93] y el de [Fuchs99] utilizan un formalismo de especificación que describe las tareas individualmente según el conocimiento de entrada, las funciones que resuelve la tarea, el conocimiento de salida, y los modelos de conocimiento de soporte para la tarea. Sin embargo, esta aproximación es demasiado restrictiva ya que supone que todos los métodos que resuelven esta tarea comparten todas estas características. Nuestra propuesta permite especificar el conocimiento de soporte para cada método que será distinto, en general, del conocimiento utilizado por otros métodos que resuelven la misma tarea.

Las características de distintos métodos relativas al conocimiento de entrada, de salida y los modelos de conocimiento adicional, pueden ser distintas incluso para métodos que resuelvan la misma tarea. Por tanto, en contraposición a otros modelos no formalizamos estas características en las descripciones de las tareas, sino en las descripciones de los métodos.

Utilizaremos la noción de estructura de tareas y métodos que se define en [Chandrasekaran90] donde la estructura de tareas identifica métodos alternativos para cada tarea y cada método puede dar lugar a distintas subtareas. En nuestro sistema la representación de estas tareas y métodos es la fuente de conocimiento independiente del dominio que se reutiliza para diseñar aplicaciones CBR. La Figura 4-5 (izda.) muestra los primeros niveles de la estructura –jerarquía *es un*– de tareas de CBROnto cuya raíz es `CBR_Task`, un concepto que representa la clase de todas las tareas CBR. Dicho concepto agrupa la información que representamos de cualquier tarea: el nombre, una descripción en lenguaje natural del objetivo principal que satisface, la relación con los distintos métodos CBR alternativos que se podrían utilizar para resolver esta tarea y la relación con el método elegido para resolverla (cuando ya se haya decidido).

Como ya hemos comentado, los requisitos de conocimiento no se representan en las tareas como proponen en [Armengol&Plaza93] ya que pueden no ser comunes y depender del método concreto. Cualquier tarea CBR concreta se representa como un individuo que es instancia directa de alguno de los conceptos de la jerarquía de tareas, y por tanto, cualquier tarea será instancia –posiblemente indirecta– del concepto `CBR_Task`. Para cada concepto de la jerarquía de tareas definimos una instancia que es el representante canónico de esta tarea. Por ejemplo, el individuo `iCBR_Task` representa la tarea de "resolver un problema y aprender de la experiencia" [Aamodt&Plaza94] y es instancia directa del concepto `CBR_Task`, la raíz de la jerarquía de tareas involucradas con la resolución de problemas. En nuestra biblioteca el único método que resuelve esa tarea es el "razonamiento basado en casos".

El conjunto de todos los individuos canónicos, y las relaciones entre ellos, representa todas las opciones de diseño, y se utiliza como punto de partida para definir ciclos de resolución de problemas en los sistemas diseñados con CBROnto. Durante el diseño de una aplicación CBR usando COLIBRI se crearán copias de estos individuos canónicos –nuevas instancias de los conceptos de CBROnto– que serán las que se configuran según las necesidades de diseño de la aplicación concreta. Esta configuración supone, por ejemplo, elegir cuál de los métodos alternativos se usará para resolver cada una de las tareas. Así cada individuo

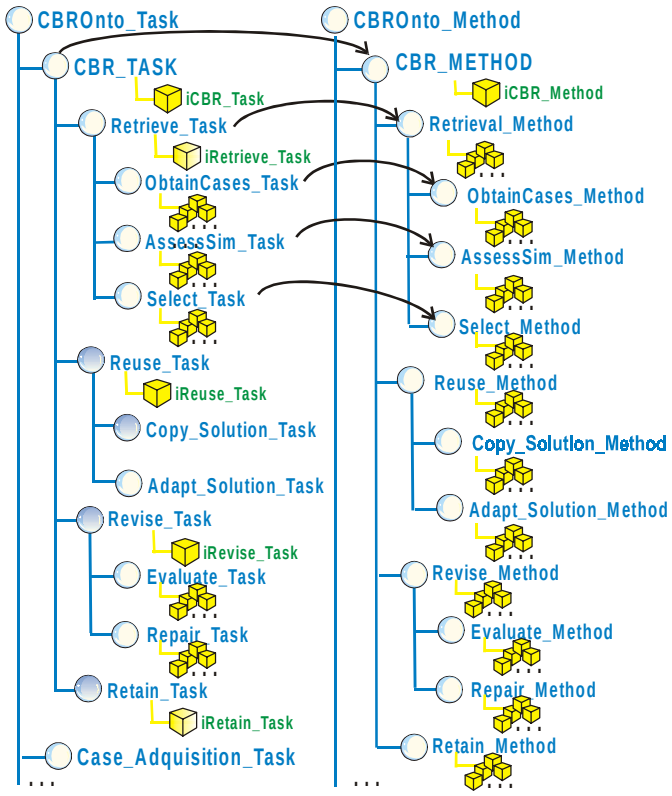


Figura 4-5. *Estructura de tareas y métodos alternativos para resolverlas de CBROnto*

tarea (la copia del individuo canónico) quedará ligado a un individuo método (copia del individuo canónico que representa al método de la biblioteca).

Clasificados bajo el concepto raíz CBR_TASK, en el primer nivel de la jerarquía encontramos los conceptos RETRIEVE_TASK, REUSE_TASK, REVISE_TASK y RETAIN_TASK. Para cada uno definimos una única instancia directa que es el individuo canónico que representa cada una de las cuatro tareas básicas de un ciclo CBR. En general, la división de una tarea en subtareas dependerá del método concreto que se utilice. En los niveles subsiguientes de la jerarquía de tareas, bajo cada concepto se definen subconceptos que agrupan a las subtareas que resulten de aplicar cualquier método de descomposición que resuelve la tarea representada por ese concepto. Por ejemplo, bajo el concepto RETRIEVE_TASK encontraremos conceptos que representan todas las subtareas que resultan de aplicar alguno de los métodos que resuelven la tarea de recuperación. La información de qué subtareas concretas se derivan de qué métodos se representa en los propios métodos. Las subtareas que provienen de los distintos métodos de recuperación se organizan en los siguientes conceptos:

- La subtarea de obtención de casos (ObtainCases_Task) se encarga de seleccionar el conjunto de casos de la biblioteca sobre el que trabajarán las siguientes subtareas. Este conjunto puede ser la base de casos completa o un subconjunto de ella filtrado de alguna forma. Esta tarea se corresponde con las subtareas *identify features* y *search* del esquema de [Aamodt&Plaza94] cuyo objetivo es el de identificar los descriptores relevantes de la consulta y, en base a ellos, buscar un conjunto de casos similares.

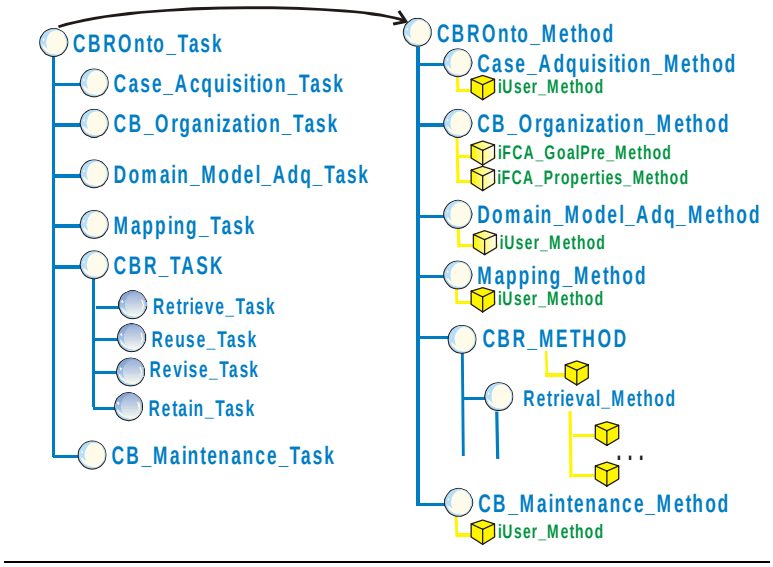


Figura 4-6. *Estructura de Tareas y Métodos de CBROnto*

- La subtarea de valoración de similitud (`AssessSim_Task`) se encarga de valorar la similitud entre el caso consulta y cada uno de los casos obtenidos por la tarea anterior. Esta tarea se corresponde con la subtarea *initially match* [Aamodt&Plaza94] cuyo objetivo es el de relacionar los descriptores del caso nuevo y los casos previos.
- La subtarea de selección (`Select_Task`) (compartida con el esquema de [Aamodt&Plaza94]) se encarga de seleccionar los casos recuperados utilizando normalmente los resultados de similitud obtenidos por la tarea anterior.

Además de la jerarquía de tareas involucradas en la resolución de problemas, la Figura 4-6 muestra que en CBROnto se han definido otras tareas –al mismo nivel que `CBR_TASK`– cuya resolución (igual que la de la tarea `CBR_TASK` para resolver problemas) se invoca externamente con una llamada al resolutor de tareas (se describe en el Capítulo 6). Ejemplos de estas tareas serían la población inicial de la base de casos (`Case_Acquisition_Task`), la adquisición del modelo del dominio (`Domain_Model_Adq_Task`), la integración del conocimiento del dominio con el conocimiento de CBROnto (`Mapping_Task`), la organización de los casos en una estructura que facilite su acceso (`CB_Organization_Task`) y el mantenimiento de la base de casos (`CB_Maintenance_Task`), que se realizará periódicamente para evitar el degradamiento del rendimiento en la resolución de problemas.

En la versión actual de CBROnto, hemos hecho más hincapié en los métodos asociados con las tareas involucradas en la resolución de problemas (subconceptos de `CBR_TASK`) por lo que los métodos asociados a las tareas adicionales delegan la resolución de las mismas al diseñador. En un futuro inmediato incluiremos nuevos métodos, por ejemplo, adquisición de casos a partir de archivos, distintas estructuras de organización de la base de casos (aparte de la organización basada en el *Análisis Formal de Conceptos* que se describe en el Apartado 5.3) o estrategias de mantenimiento de la base de casos con estrategias de olvido.

3.2 Ontología de métodos CBR

Hablaremos de ontología de métodos para referirnos tanto a las definiciones de los elementos que se utilizan para representar a los métodos –que definen los términos del lenguaje de

representación– como a las definiciones de los métodos en sí mismos incluyendo la terminología necesaria para organizarlos por tareas, y los individuos que representan propiamente a los métodos –que constituyen la biblioteca de PSMs de CBROnto.

Para la representación de los métodos, CBROnto define una jerarquía de métodos que es paralela a la jerarquía de tareas y cuya raíz es el concepto CBR_METHOD que representa la clase de los métodos CBR, es decir, CBR_METHOD agrupa la información común de todos los PSMs que se representarán como instancias –directas o no– suyas.

Para cada método representamos su nombre, una descripción informal, su competencia –la tarea que resuelve–, la salida que genera, una descripción de los modelos de conocimiento que necesita –requisitos– tanto las entradas externas como los que forman parte del modelo del dominio, y su especificación operacional, que hace referencia a una función Lisp. Distinguimos entre dos tipos de métodos:

- Los métodos de *descomposición* resuelven una tarea dividiéndola en subtareas que deben ser resueltas en un cierto orden.
- Los métodos de *resolución* son los que directamente resuelven las tareas.

Los PSMs de la biblioteca se representan como instancias de alguno de los conceptos de la jerarquía de métodos (subconceptos de CBR_METHOD). El concepto CBR_METHOD tiene una única instancia directa iCBR_METHOD que es el representante canónico del método "razonamiento basado en casos", un método de descomposición que resuelve la tarea (iCBR_Task) "resolver un problema y aprender de la experiencia". El siguiente nivel de la jerarquía cuenta con cuatro subconceptos del concepto raíz: RETRIEVAL_METHOD, REUSE_METHOD, REVISE_METHOD y RETAIN_METHOD (Figura 4-5). El paralelismo entre las jerarquías de tareas y métodos permite formalizar los compromisos ontológicos mediante restricciones que relacionan los conceptos de ambas jerarquías. Por ejemplo, cada instancia de RETRIEVAL_METHOD es un método que resuelve una tarea representada por una instancia de RETRIEVE_TASK.

```
(defconcept Retrieve_Task :is-primitive
  (:and CBR_TASK
       (:filled-by task_name "Retrieve")
       (:all task_method Retrieval_Method))
:annotations ((documentation "the concept representing the task of retrieving
cases from the case base. ")))
```

El role task_method relaciona una tarea con el método, de todos los alternativos, que se haya elegido para resolverla en una aplicación concreta. Como veremos en el Capítulo 6, el diseño de una aplicación con COLIBRI/CBROnto se basa en establecer estos enlaces mediante los cuales el diseñador expresa qué método utilizar de entre todos los métodos alternativos que son capaces de resolver cada tarea.

Adicionalmente cada concepto de la jerarquía de métodos puede tener subconceptos cuyas instancias representan métodos que resuelven alguna subtarea de las producidas por ese método. Por ejemplo, bajo el concepto RETRIEVAL_METHOD podemos encontrar subconceptos –como OBTAIN_CASES_METHOD o ASSESSSIM_METHOD– cuyas instancias representan métodos de la biblioteca que resuelven alguna de las subtareas producidas por alguno de los métodos de recuperación.

3.2.1 El lenguaje de representación de métodos

Para que un diseñador pueda reutilizar métodos de una biblioteca debe ser capaz de encontrar y entender estos métodos. Por esta razón cada método debe describirse usando un cierto

lenguaje de representación que permita especificar, entre otras cosas, la funcionalidad conseguida, es decir, la tarea que resuelve, cuáles son los datos de entrada y de salida y los requisitos de conocimiento adicional del método.

En nuestra arquitectura, y con el objetivo de mantener la homogeneidad y consistencia de la representación de conocimiento CBR explícito, hemos decidido utilizar terminología incluida en CBROnto junto con el lenguaje de descripciones de LOOM, como lenguaje de representación de métodos. Adicionalmente, el uso de especificaciones con el IDL de CORBA facilitará el intercambio de información con las funciones de la interfaz gráfica de COLIBRI. En la literatura existen distintos lenguajes de especificación de métodos de resolución de problemas, tales como CML [Schreiber *et al.* 94], UPML [Fensel *et al.* 98a] y Noos [Arcos97]. Aunque algunos de estos lenguajes han influido en nuestro trabajo, no hemos realizado un estudio detallado al respecto. En [Gennari *et al.* 98] se definen cuatro dimensiones para caracterizar un lenguaje de descripción de métodos:

- *Grado de formalismo del lenguaje.* La información del método se puede expresar con varios niveles de formalismo desde el uso de expresiones axiomáticas formales hasta texto libre.

  Nuestra aproximación usando LOOM hace que nuestra propuesta global sea inherentemente formal y precisa. Como veremos esto nos permite verificar formalmente si los requisitos de aplicación de un métodos son satisfechos o no por una cierto modelo del dominio o un conjunto de casos. De todas formas, como documentación y para facilitar la tarea del diseñador de aplicaciones CBR, también incluimos la opción de añadir descripciones en lenguaje natural en la representación de los métodos.

- *Cómo vs qué.* El propio lenguaje puede describir qué cosas hace el método sin detallar aspectos del flujo de control, o puede describir explícitamente el flujo de control que especifica cómo se utiliza el conocimiento para resolver el problema.

  Nuestro lenguaje describe *qué* cosas hace el método sin detallar aspectos de su flujo de control. Aunque en CBROnto todos los métodos están implementados (en Lisp/LOOM), estas implementaciones no se caracterizan formalmente como parte de la descripción de los métodos.

- *Objetivos del lenguaje.* En general, el lenguaje proporciona información que ayuda a seleccionar un método de la biblioteca pero además puede ayudar a aplicar el método a una base de conocimiento en particular, especificando qué cambios se requerirían para poder aplicar el método en función de los requisitos de conocimiento de dicho método.

  El objetivo deseado para nuestro lenguaje es, principalmente, el de proporcionar suficiente información para que el diseñador de una aplicación CBR pueda seleccionar un método de la biblioteca. Nuestro lenguaje facilita que el sistema de DLs pueda comprobar formalmente si un método es aplicable en base a sus requisitos de conocimiento y dar una explicación razonada al respecto.

- *Descripciones a nivel de conocimiento o a nivel simbólico.* Las descripciones abstractas, a nivel de conocimiento, de un método son más fácilmente reutilizables. Sin embargo, si el lenguaje de descripción de métodos está ligado a la implementación a nivel simbólico es más sencillo adaptar el método para su reutilización.

  Nuestra aproximación liga el lenguaje de descripción de métodos (en LOOM) a la implementación a nivel simbólico (en Lisp). En nuestro caso la reutilización de mé-

todos permite únicamente adaptaciones simples de los métodos basadas en la configuración de sus requisitos y no en transformaciones sobre su código.

En definitiva, queremos que nuestro lenguaje describa los métodos basándose en el uso de ontologías formales, abstrayendo detalles de control, flujo de datos y otros aspectos dinámicos de los PSMs. Nuestra aproximación se concentra en la competencia y en los requisitos de conocimiento de un método, en vez de en los detalles internos de control, lo que permite un tipo de reutilización de caja negra. Esta aproximación es distinta a la planteada, por ejemplo, en CML [Schreiber *et al.* 94] y en Noos [Arcos97], donde se requiere especificar gran parte del proceso interno de razonamiento de un PSM. En nuestras descripciones de métodos los detalles de control no se formalizan como parte de las representaciones con las que se razona, sino que cada descripción de un método de resolución hace referencia a la función Lisp que implementa ese método, lo que permite reutilizar y parametrizar el código pero no hacer modificaciones en él.

La representación explícita de los requisitos de conocimiento de un método permite que se pueda comprobar dinámicamente su aplicabilidad para un cierto modelo del dominio y conjunto de casos. Uno de los objetivos de nuestro lenguaje de descripción de métodos es que facilite la tarea de integración entre la terminología del dominio y la del método, es decir, que la representación de un método nos permita determinar su aplicabilidad respecto a una base de conocimiento terminológico del dominio. Además, el sistema podrá sugerir ciertas modificaciones en el conocimiento para que el método sea aplicable, por ejemplo, se puede sugerir la clasificación de algún término del dominio bajo algún término de CBROnto.

Una ventaja, y la diferencia principal, de nuestro trabajo con otros relativos a ontologías y bibliotecas de métodos reutilizables es que sólo estamos interesados en los métodos específicos de las tareas CBR y, por tanto, no es necesario tener en cuenta el mantenimiento de un compromiso entre la aplicabilidad vs. generalidad que surge al desarrollar métodos demasiado generales y poco aplicables [Beys *et al.* 96] [Fensel *et al.* 97]. Los métodos propuestos en nuestro sistema son fácilmente aplicables y reutilizables en el contexto que nos interesa, las tareas involucradas en el CBR. Además la labor de organizar e indexar los métodos se simplifica utilizando esta organización por tareas.

3.2.1.1 El lenguaje de representación de métodos de CBROnto

En CBROnto cada método se representa como una instancia del concepto CBR_METHOD, y es la propia definición LOOM de ese concepto (ver Apéndice B) la que restringe y caracteriza a sus instancias. En este sentido, el lenguaje utilizado para formalizar los métodos concretos de la biblioteca utiliza los constructores del lenguaje de definición de individuos de LOOM, restringidos a la definición de instancias de CBR_METHOD. Para proporcionar una visión general de la representación de los métodos, y para facilitar la comprensión a los lectores que no estén familiarizados con la sintaxis LOOM, incluimos la sintaxis EBNF del lenguaje de definición de métodos de CBROnto, donde los términos de CBROnto aparecen en cursiva:

```
<CBR-Method> ::= <CBROnto-Method instance>
<CBROnto-Method instance> ::= <Resolution_Method instance> |
                              <Decomposition_Method instance>
<Resolution_Method instance> ::=  <Method-description>
<Decomposition_Method instance> ::= <Method-description> {<subtask>}+
<Method-description> ::= <name> <informal-description> {<competence>}+
                         [<application-requirements>] [<input>] {<output>}*
                         <operational-specification>
```

```
<name> ::= method_name <String>
<informal-description> ::= method_informal_description <String>
<competence> ::= competence <CBR-Task-instance>
<application-requirements> ::= method_application_requirements
                                <Application_Requirements instance>
<input> ::= method_input <Method_Input instance>
<output> ::= method_output <Method_Output instance>
<operational-specification>::= operational_specification
                                <Method_Function instance>
<subtask> ::= subtask <CBR_Task instance>
<CBR_Task instance> ::= iRetrieve_task | iReuse_task | iRevise_task | iRetain_task |
.....
<Method_Input instance> ::=[<knowledge-requirements>]
                            [<parameter-requirements>]
                            [<design-requirements>]
                            [<sequence-requirements>]
<knowledge-requirements>::= knowledge_requirements
                            <Knowledge_Requirements instance>
<parameter-requirements>::= parameter_requirements
                             <Parameter_Requirements instance>
<design-requirements> ::= design_requirements
                          <Design_Requirements instance>
<sequence-requirements> ::= sequence_requirements
                           <Sequence_Requirements instance>
```

Por tanto, cada método de la biblioteca es un individuo –instancia de CBR_METHOD– que se describe utilizando las siguientes relaciones:

- method_name relaciona cada método con su nombre.
- method_informal_description relaciona cada método con una descripción en lenguaje natural de la especificación informal del método, con propósito de documentar el método.
- competence relaciona cada método con la instancia de CBR_TASK que representa la competencia del método –la descripción de qué hace el método.
- method_application_requirements relaciona cada método con sus *requisitos de aplicabilidad* que representan el conocimiento necesario para que el método sea aplicable (ver Apartado 3.2.1.3).
- method_input relaciona cada método con un individuo que aúna todos los requisitos de entrada del método, es decir, *de diseño*, de *secuencia*, de *conocimiento* y *paramétricos* (ver Apartado 3.2.1.3).
- method_output relaciona cada método con un individuo que representa los resultados de salida del método. Este individuo se usa para comprobar la corrección del método cuando aparece en una secuencia de resolución M_1, M_2, ..., M_n en la que los requisitos de secuencia del método M_i deben ser satisfechos por el individuo que representa la salida del método M_{i-1}.

- `operational_specification` relaciona cada método con su especificación operacional determinada por la relación con una instancia de `Method_Function`, que representa la función Lisp que implementa el comportamiento del método.

Para distinguir entre los métodos de descomposición y de resolución se utiliza una partición exhaustiva sobre el concepto `CBR_METHOD` que indica que todas sus instancias deben pertenecer a uno de los dos conceptos `DECOMPOSITION_METHOD` o `RESOLUTION_METHOD`. Los apartados siguientes describen cómo se organizan los métodos de CBROnto en torno a las tareas que resuelven y algunos aspectos relativos a los requisitos de conocimiento de los métodos.

3.2.1.2 Organización de los métodos en CBROnto

Ya hemos comentado que en nuestra aproximación los métodos se definen de forma dependiente de la tarea que resuelven. Esta característica, junto con la granularidad gruesa de los métodos de CBROnto, facilita la reutilización de los métodos en el contexto de las tareas CBR, y hace que sea adecuada una organización clásica de los métodos de forma estática en torno a las tareas que resuelven. Cada método se relaciona a través de la relación `competence` con los individuos canónicos de las tareas que resuelve –ya que, en general, un método puede resolver varias tareas.

En nuestra estructura cada tarea se relaciona con una colección de métodos de resolución alternativos. Esta relación no se representa de forma explícita sino que para obtener la colección de métodos se utiliza la relación inversa de la competencia de los métodos (relación `competence`). La elección de uno u otro método para resolver dicha tarea puede depender de varios factores, principalmente de las características del conocimiento disponible, que determinarán la aplicabilidad del método en una cierta situación o contexto. El comportamiento a nivel de conocimiento de los métodos que resuelven una misma tarea es similar, por ejemplo, todos los métodos asociados a la tarea de recuperación obtienen el/los casos más similares a una consulta dada. Las diferencias entre los métodos se refieren, por ejemplo, al modo de valorar la similitud entre los casos, a las restricciones de aplicabilidad de cada método, a sus requisitos de conocimiento o al nivel de intervención del usuario.

La competencia de un método es una descripción declarativa del comportamiento del método. Cuando se manejan bibliotecas genéricas, la competencia de los métodos es un factor muy importante a tener en cuenta ya que el proceso de encontrar un método adecuado para resolver unos ciertos objetivos no es en absoluto trivial.

Aunque existen aproximaciones más complejas (ver por ejemplo, la descrita para el sistema EXPECT [Gil&Melz96]), nosotros hemos elegido una representación simple de la competencia de los métodos basada en las tareas CBR que están claramente definidas y representadas explícitamente.

Dada una tarea, el resolutor selecciona los métodos aplicables teniendo en cuenta su competencia –relación `competence` entre el método y la tarea– y si el conocimiento disponible satisface los requisitos de aplicabilidad del método. De los métodos seleccionados por el resolutor de tareas de COLIBRI el diseñador elige el método que se utilizará para resolver esa tarea en la aplicación. Es decir, para diseñar una aplicación usando COLIBRI el diseñador debe elegir qué tareas se resolverán en cada ciclo CBR (por ejemplo, puede diseñar una aplicación CBR de sólo recuperación). El sistema –basándose en la competencia de los métodos– ofrecerá al diseñador los métodos que existen para cada tarea a resolver. De los métodos ofrecidos por el sistema el diseñador elige y *configura* los que serán ofrecidos a los usuarios finales de la aplicación. La configuración se realiza a través de los requisitos especificados para los métodos.

3.2.1.3 Requisitos de los métodos

Para configurar un método de la biblioteca de CBROnto el diseñador debe proporcionar cierto conocimiento –requerido por el método. Este es el mecanismo mediante el cual el diseñador puede incluir los elementos específicos de la aplicación que completan los métodos genéricos de CBROnto.

Los *requisitos de entrada* de los métodos se refieren a cualquiera de sus entradas externas. Dependiendo de cómo y quién los especifica distinguimos entre los siguientes:

- Los *requisitos de conocimiento* del método se refieren al conocimiento en forma de individuos, conceptos o relaciones que se integran dentro del modelo del dominio. Estos elementos pueden ser de tipo procedimental –como las funciones de similitud, los criterios lógicos de relevancia, o las estrategias de adaptación y revisión– o terminológico, como la clasificación de ciertos términos del dominio bajo ciertos términos de CBROnto. En ambos casos, su objetivo es completar los métodos genéricos y particularizarlos para su uso en una aplicación concreta. Los requisitos de conocimiento añaden términos o instancias a la base de conocimiento de la aplicación. Los métodos están definidos de forma que acceden al conocimiento concreto que esté definido en cada momento. Si estos elementos no se definen el método utiliza recursos genéricos, independientes del dominio y de la aplicación de los que podemos esperar "peores" resultados. Los requisitos de conocimiento plantean oportunidades de adquisición de conocimiento del dominio, es decir, indican al diseñador qué elementos es conveniente añadir como parte del conocimiento explícito específico de la aplicación concreta.
- Los *requisitos de diseño* del método son entradas externas del método (parámetros) que son especificados por el diseñador. Como estas entradas se fijan en la fase de diseño del sistema serán las mismas para todas las ejecuciones del método.
- Los *requisitos paramétricos* del método son las entradas del método que varían de una a otra ejecución y que son especificadas por el usuario final y no por el diseñador. Por ejemplo, la consulta de un ciclo CBR.
- Los *requisitos de secuencia* son las entradas que el método espera recibir del método que resuelve la tarea anterior en una secuencia de resolución de tareas. Por ejemplo, para que el método de adaptación pueda funcionar se requiere que el método que resuelve la tarea anterior (la recuperación) devuelva el conjunto de casos seleccionados.

Durante el diseño de una aplicación CBR, el diseñador configurará ciertos métodos de la biblioteca de PSMs CBROnto, especificando sus requisitos –de conocimiento y de diseño– de forma adecuada. La intervención del usuario final en la aplicación consiste en la configuración de los requisitos paramétricos de los métodos que resuelven las tareas.

Los *requisitos de aplicabilidad* del método determinan las características mínimas que debe cumplir una situación o contexto para que se pueda aplicar un método. Un contexto de aplicación se describe según las características del modelo del dominio y del conjunto de casos. Este tipo de requisitos se utilizan para determinar la aplicabilidad respecto de un contexto concreto y pueden inhibir o recomendar la aplicación de un método en dicho contexto. En general, los requisitos de aplicabilidad se pueden solapar con cualquiera de los componentes de los requisitos de entrada, es decir, compartir elementos. Este mecanismo permite expresar la obligatoriedad de alguno de los requisitos de entrada, ya que todos los requisitos de aplicabilidad se deben satisfacer previamente a la aplicación del método.

Como veremos en detalle para cada uno de los métodos de CBROnto, que se describen en el Capítulo 5 y se resumen en el Apéndice C, los requisitos de aplicabilidad hacen referencia a ciertos aspectos del modelo de conocimiento del dominio –por ejemplo, la profundidad y anchura de las jerarquías de conceptos y relaciones–, aspectos de la base de casos –como el número de casos y los tipos de casos que aparecen, por ejemplo, casos con solución. Los requisitos de aplicabilidad también se refieren a otros aspectos como el tipo de conocimiento del dominio que necesita el método en función de los papeles que juegan sus términos. El papel que juega un término del dominio depende de cómo esté clasificado bajo los términos de CBROnto. Por ejemplo, un método de recuperación basado en la satisfacción de objetivos requiere que existan términos del dominio que jueguen el papel de objetivos, es decir, que estén clasificados bajo el concepto `Goal` de CBROnto, así como que los casos describan los objetivos que llevan a cabo.

La representación explícita y declarativa de los requisitos de aplicabilidad de los métodos hace que sea posible razonar con ellos y comprobar si un método es o no aplicable para una cierta situación o contexto. Como se describe en el Capítulo 6, COLIBRI aprovecha los mecanismos de LOOM para razonar con las descripciones de los requisitos y poder comprobar si el contexto externo de aplicación cumple o no los requisitos de aplicabilidad del método. De esta forma, los requisitos de un PSM que no están disponibles representan una oportunidad de aprendizaje, sobre todo los requisitos de conocimiento que guían al diseñador para que completen el modelo de conocimiento del dominio con conocimiento específico de la aplicación (definición de medidas de similitud, criterios de relevancia, estrategias de adaptación, etc.).

Igual que en COLIBRI, el sistema EXPECT también propone mecanismos para encontrar errores y falta de conocimiento, así como para sugerir remedios para repararlos basados en las capacidades de razonamiento de LOOM. Esta idea estaba también presente en la metodología *Generic Task* [Chandrasekaran86] que resalta el hecho de que la estructura de tareas y métodos asociada con la resolución de un problema ayuda en el proceso de adquisición de conocimiento, al identificar y representar explícitamente los requisitos de conocimiento de los métodos.

3.3 El método CBR

El comportamiento del método CBR se ha explicado de forma detallada en el Capítulo 2 por lo que, en este apartado, nos centramos en la descripción de su formalización en CBROnto. Como hemos descrito en el apartado anterior, cada método se representa como un individuo que es instancia del concepto `CBR_METHOD`. El individuo `iCBR_Method` es la instancia canónica que representa al método de razonamiento basado en casos (el Apéndice B incluye su definición en LOOM).

Observamos en la Figura 4-7 que `iCBR_Method` es un método de descomposición –es decir, una instancia de `Decomposition_Method`– que resuelve la tarea `iCBR_task`, dividiendola en las cuatro subtareas citadas: recuperación –`Retrieve_Task` (a través de su instancia canónica `iRetrieve_task`)–, adaptación –`Reuse_Task` (`iReuse_task`)–, revisión –`Revise_Task` (`iRevise_task`)– y aprendizaje –`Retain_Task` (`iRetain_Task`).

El método CBR genera como salida explícita el individuo `iCBR_Output` con los atributos siguientes que tendrán valores o no dependiendo de las tareas que se resuelvan:

- `retrieved_cases` contiene la lista ordenada de casos recuperados al resolver la tarea de recuperación.

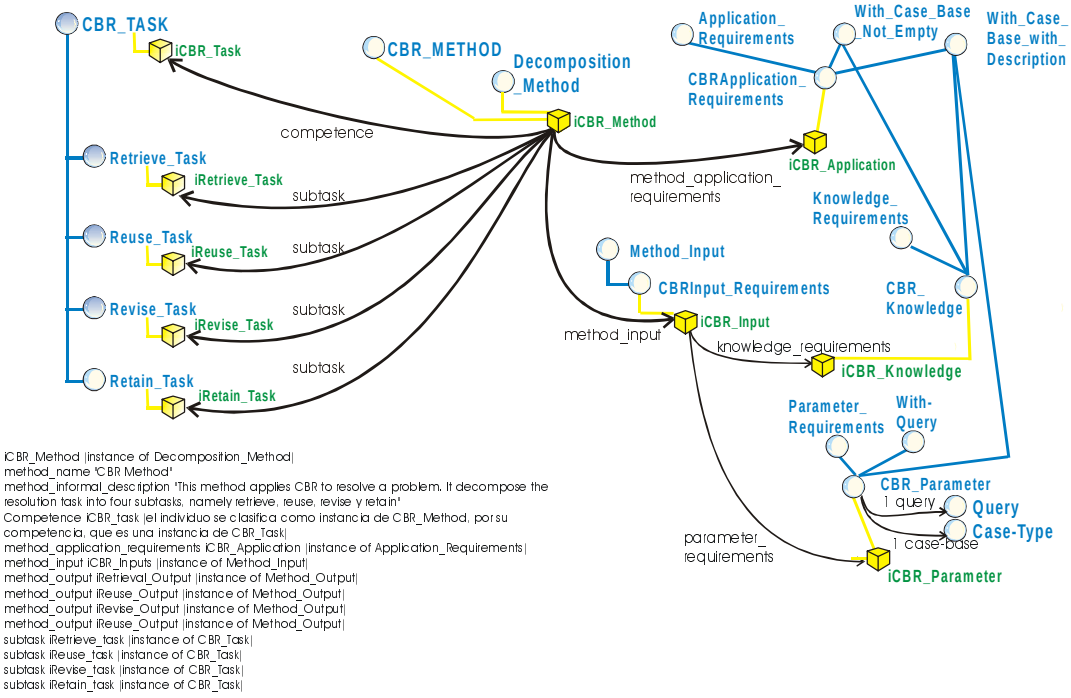


Figura 4-7. *Representación del método CBR en CBROnto*

- `reused_case` contiene el caso adaptado. Este atributo sólo tendrá un valor cuando se resuelve la tarea `iReuse_Task`..
- `revised_case` contiene el caso revisado cuando se resuelve la tarea `iRevise_Task`.
- `retained_case` contiene el caso añadido a la base de conocimiento si al resolver la tarea `iRetain_Task` se añade el caso.

Además el método CBR puede generar más resultados en forma de conocimiento que se añade al modelo de dominio (en el aprendizaje). Pero no se refleja como salida explícita.

Los subapartados siguientes describen los requisitos del método CBR así como las subtareas derivadas de la aplicación de este método. Cada una de las subtareas se resuelve utilizando alguno de los métodos cuya competencia sea adecuada. Como la única tarea obligatoria es la recuperación de casos, el resto de las tareas pueden resolverse con un método genérico que representa la no resolución (`iDo_Nothing_Method`).

3.3.1 Requisitos del método CBR

Para poder aplicar el método CBR se requiere al menos disponer de una base de casos. Por tanto, hemos identificado como requisito de aplicabilidad la existencia de una base de casos, con algún caso y en la que los casos tienen descripción. La Figura 4-7 esquematiza la representación de los requisitos del método CBR.

Como parte de los requisitos de entrada del método se identifican sus requisitos de conocimiento y sus requisitos paramétricos. Los requisitos de conocimiento coinciden con los de aplicabilidad, es decir, se requiere la existencia de una base de casos. Como requisitos paramétricos, por un lado el usuario final especificará la consulta, que debe ser una instancia de

`Query-Type`, y por otro lado, la base de casos en la que buscaremos. Como veremos en el Capítulo 6 las relaciones `query` y `case-base` forman parte de la descripción del contexto en el que se aplicará el método CBR. La representación explícita de los requisitos de aplicabilidad de los métodos junto con los mecanismos de razonamiento de las DLs, permiten determinar si un determinado método es aplicable en un cierto contexto descrito por el conocimiento del dominio disponible. Para ello se razona con una descripción explícita del contexto y se utilizan ciertos conceptos predefinidos de CBROnto, que permiten cualificar el conocimiento disponible en el sistema CBR. Por ejemplo, los conceptos `With_Query`, `With_CaseBase_Not_Empty` y `With_CaseBase_with_Description` que aparecen en la descripción de los requisitos del método CBR:

```
(defconcept With_Query :is (:the query Query-Type))
(defconcept With_CaseBase_with_Description :is (:and
                (:the case-base Case_With_Description)))
(defconcept With_CaseBase_not_Empty :is (:and (> case-base-size 0)))
```

3.3.2 Tarea y métodos de recuperación

Un sistema CBR es un sistema de razonamiento cuyos procesos se basan en la recuperación, y posterior adaptación, de episodios previamente resueltos. En la literatura existen muchas aproximaciones y propuestas de métodos que resuelven la tarea de recuperación. De hecho el proceso de recuperación es uno de los que ha recibido más interés dentro de la comunidad de CBR, debido a que cualquier sistema de CBR tiene un modulo de recuperación, aunque no todos los sistemas llevan a cabo el resto de las tareas. La revisión y la adaptación a menudo se simplifican e incluso desaparecen en muchas de las propuestas. Además, el éxito de un determinado sistema depende de forma crítica de la recuperación eficiente del caso adecuado en el momento preciso [Smyth&McKenna99]. La recuperación involucra subtareas como la valoración de la similitud y la selección de casos que son tareas básicas en cualquier sistema de CBR ya que el resto de las tareas sólo tendrán éxito si los casos previos considerados son los apropiados.

En general, para determinar que dos características de los casos son *similares*, cualquier sistema CBR debería disponer de un mínimo y "suficiente" modelo del dominio, que puede depender únicamente del tipo de la característica –10 es más similar a 20 si el tipo es [0..1000] que si el tipo es [0..20]– o ser idiosincrásico –15 y 30 son similares cuando hablamos de resultados de tenis pero son valores muy distintos si representan resultados de baloncesto. En este caso, para valorar la similitud tendremos que disponer explícitamente del conocimiento de que en baloncesto se cuenta generalmente de 2 en 2 y que una diferencia de 15 puntos representa realmente 15 puntos, pero en tenis una diferencia de 15 puntos representa sólo 1 punto. El proceso de valoración de la similitud utilizará este conocimiento cuando disponga de él, pero si no, podría utilizar una medida de similitud general menos precisa, por ejemplo, basada en la diferencia de los valores si son numéricos o comprobando si son iguales para valores simbólicos.

La aproximación KI-CBR tomada en CBROnto nos ha llevado a estudiar distintas posibilidades que ofrece el uso del conocimiento general del dominio durante la resolución de la tarea de recuperación de casos. Por un lado, como vimos en el Capítulo 2, la jerarquía conceptual del modelo del dominio contiene conocimiento sobre la similitud de sus objetos. Además, los mecanismos de clasificación y reconocimiento de instancias de las DLs, permiten definir métodos de recuperación que utilizan la jerarquía conceptual del modelo del do-

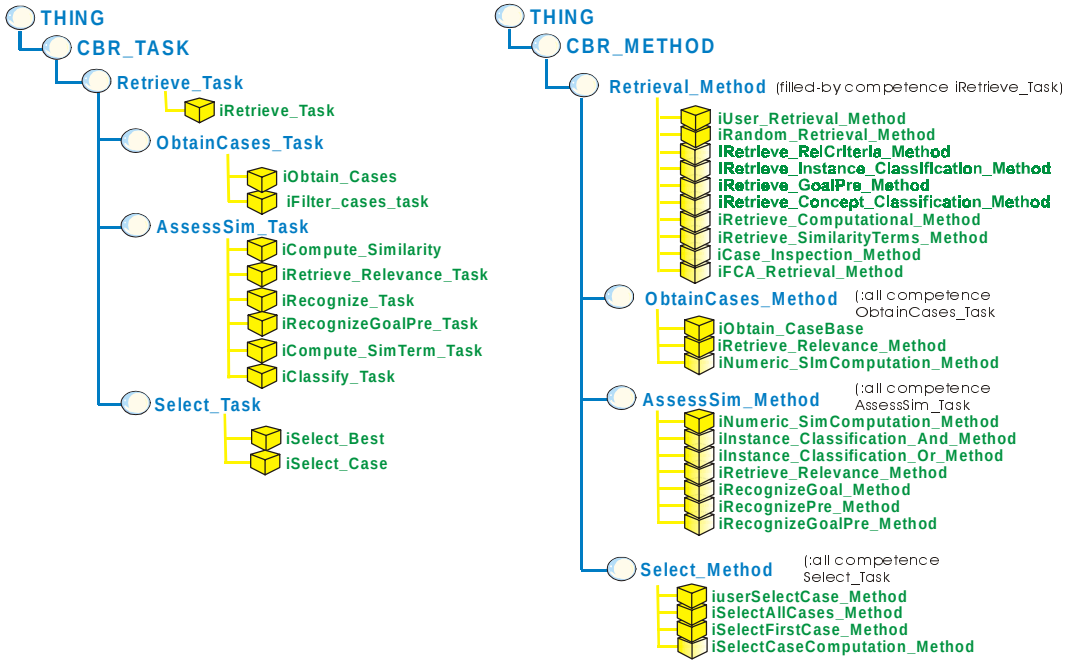


Figura 4-8. *Tareas y métodos de recuperación*

minio como una estructura inicial de organización de casos, que se puede completar con conceptos índices específicos de cada aplicación concreta [Salotti&Ventos98].

CBROnto ofrece varios métodos alternativos para resolver la tarea de recuperación de casos. La Figura 4-8 muestra la jerarquía de clasificación conceptual correspondiente a las tareas y métodos de recuperación de CBROnto, así como los individuos canónicos, donde las aristas representan relaciones *es un* entre conceptos o entre individuos y conceptos. Cada método de recuperación se representa como una instancia directa del concepto RETRIEVAL_METHOD. Además, los subconceptos de RETRIEVAL_METHOD –`ObtainCases_Method`, `AssessSim_Method` y `Select_Method`– representan los métodos que resuelven las subtareas que se derivan de aplicar los métodos de recuperación. El sistema infiere la pertenencia de un individuo método a un concepto de la taxonomía de métodos según la competencia asertada.

La Figura 4-9 representa la estructura relacional definida por las instancias canónicas de los conceptos que representan las tareas y los métodos de recuperación. La Figura 4-9 esquematiza la estructura global resultante de la formalización del conjunto de métodos predefinidos de CBROnto. Cada uno de los métodos se relaciona a través de la relación de competencia (`competence`) con un individuo que representa la tarea que resuelve. Se observa también que la competencia de los métodos no es única en general, es decir, algunos métodos son adecuados para resolver varias tareas[1].

La relación `task_method` relaciona una tarea con el método elegido para su resolución, que debe ser uno de los métodos cuya competencia (relación `competence` entre el método y la tarea) sea adecuada para resolver esa tarea. Aunque en CBROnto algunas de las subtareas tienen ligaduras predefinidas –mediante `task_method`– con un único método que las resuel-

[1] Realmente son distintos individuos, pero si son instancias del mismo concepto tarea puede considerarse que la tarea (conceptualmente) es única.

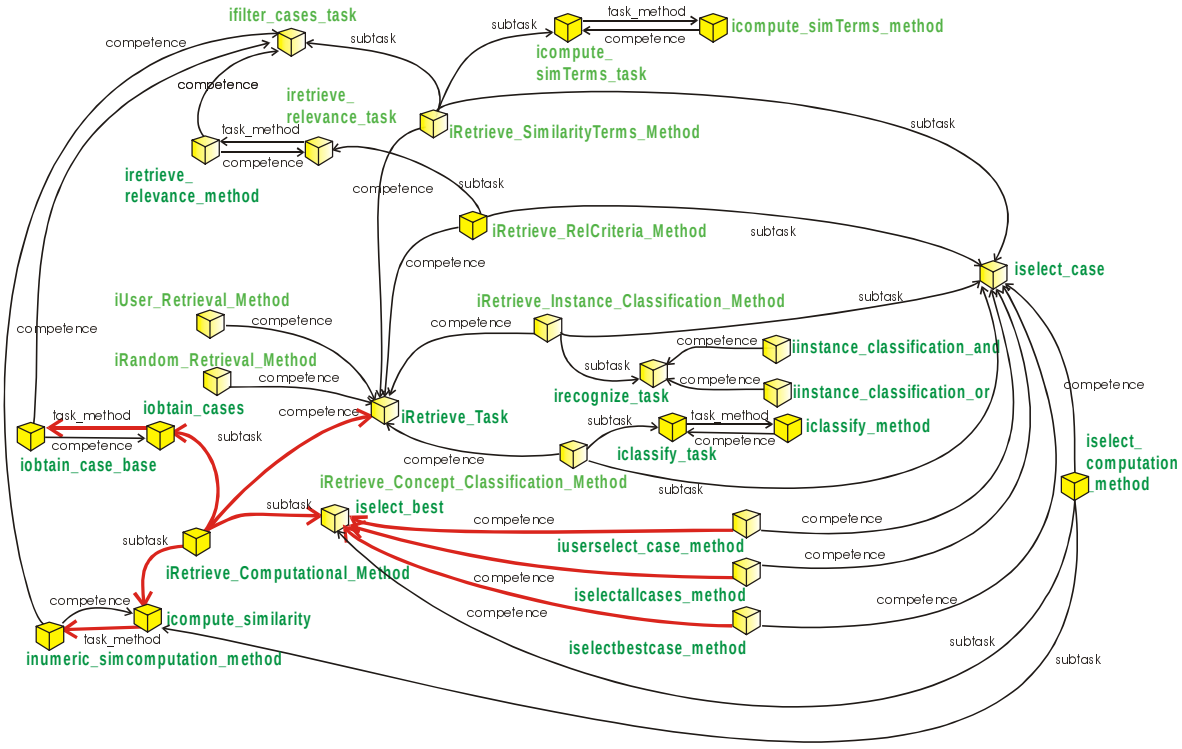


***Figura 4-9**. Estructura relacional de las tareas y métodos de recuperación*

ven, en general, para la mayoría de las tareas, esta ligadura no está predefinida. En general para cada tarea existen en la biblioteca varios métodos cuya competencia es adecuada para resolverla. En este caso el diseñador tendrá que elegir uno de ellos –se establecerá la relación `task_method`– durante el diseño de una nueva aplicación de entre los métodos con la competencia adecuada.

En la Figura 4-9 hemos resaltado los enlaces asociados con uno de los métodos de recuperación de CBROnto. El método de recuperación por cómputo de similitud (`iRetrieve_Computational_Method`) tiene como competencia el individuo `iRetrieve_Task`, es decir, el método es uno de los candidatos a resolver la tarea pero no el elegido (de momento) puesto que no hay un enlace `task_method` entre la tarea y dicho método. Se observa que es un método de descomposición cuya aplicación deriva tres subtareas que deben ser resueltas a su vez (el control entre las tareas se define en la especificación operacional del método): obtener el conjunto de candidatos iniciales (`iobtain_cases`), computar la similitud de la consulta con cada uno de ellos (`icompute_similarity`) y seleccionar el mejor (`iselect_best`). Las tareas `iobtain_cases` e `icompute_simililarity` tienen sendos métodos elegidos (enlaces `task_method`) que son de resolución porque no derivan nuevas subtareas. Para resolver la tarea `iselect_best` el sistema accede a los métodos con una competencia adecuada: `iuserselect_case_method`, `iselectallcases_method` e `iselectbestcase_method`, de los que el diseñador elegirá alguno.

Como se puede observar en la Figura 4-8, cada concepto de la taxonomía de tareas puede agrupar a varios individuos canónicos de la tarea representada por el concepto. El uso de distintos individuos permite afinar más en el proceso de organización de los métodos en

torno a las tareas que resuelven. Cada individuo método estará ligado a través de la relación de competencia con algún individuo tarea que representará la tarea que resuelve el método. El diseñador elegirá uno de entre los métodos que tengan como competencia directamente este individuo, si hay alguno, y si no entre los métodos que tengan como competencia a alguno de sus hermanos, es decir, cualquiera de las instancias del mismo concepto tarea. Cada elección de un método (de los que tenga como competencia esa instancia) se traduce en un aserto de la relación `task_method` ($\rightarrow_{task\_method}$) entre la tarea y el método.

En concreto para la tarea de recuperación, cuyo representante canónico es el individuo `iRetrieve_Task`, la inversa de la relación de competencia permite encontrar varios métodos alternativos que se pueden usar para resolverla, de los que uno de ellos será elegido asertando la relación `task_method`.

Enunciamos aquí los métodos de recuperación, aunque postponemos la descripción del comportamiento de los mismos al Capítulo 5 de esta memoria:

- Método de recuperación por el usuario. Representado por el individuo `iUser_Retrieval_Method`, es un método de resolución que delega en el usuario la tarea de recuperación de casos.
- Método de recuperación aleatoria. Representado por el individuo `iRandom_Retrieval_Method`, es un método de resolución que selecciona aleatoriamente un caso.
- Método de recuperación por cómputo de similitud numérica que se basa en computar la similitud entre la consulta y cada individuo de la base de casos. Se representa con el individuo `iRetrieve_Computational_Method`.
- Método de recuperación por reconocimiento de instancias que se basa en la recuperación de casos clasificados *cerca* de un individuo que representa la consulta actual. El individuo canónico que representa a este método es `iRetrieve_instance_Classification_Method.` Como veremos en el Capítulo 5, el comportamiento de este método varía dependiendo de la estructura conceptual sobre la que se trabaje.
- Método de recuperación por reconocimiento de instancias sobre el retículo de conceptos formales de objetivos y precondiciones obtenido al aplicar el Análisis Formal de Conceptos. Está representado por el individuo `iRetrieve_GoalPre_Method`.
- Método de recuperación exacta o inspección de casos por complección de consultas usando las reglas de dependencia resultantes de aplicar el Análisis Formal de Conceptos. Está representado por el individuo `iCase_Inspection_Method.`
- Método de recuperación por reconocimiento de instancias sobre el retículo de conceptos formales sin aplicar las reglas de dependencia extraídas al aplicar el Análisis Formal de Conceptos. Se representa con el individuo `iFCA_Retrieval_Method`.
- Método de recuperación por clasificación de conceptos. Se basa en la recuperación de casos clasificados *cerca* de un concepto que representa la consulta actual. Está representado por el individuo `iRetrieve_Concept_Classification_Method`.
- Método de recuperación por términos de similitud. Representado por el individuo `iRetrieve_SimilarityTerms_Method`.
- Método de recuperación por criterios de relevancia. Representado por el individuo `iRetrieve_RelCriteria_Method`.

3.3.3 Tarea y métodos de adaptación

Después de resolver la tarea de recuperación, obtenemos uno o varios casos similares a una consulta dada. El objetivo de la tarea de adaptación es adaptar el mejor caso recuperado para que satisfaga los requisitos especificados por la consulta.

Dentro de la comunidad de CBR la tarea de adaptación define una línea de investigación muy activa. Representa un reto para cualquier aplicación CBR ya que es la tarea que presenta un mayor grado de dificultad y una necesidad de conocimiento y métodos que son específicos del dominio y de la aplicación. Aunque este hecho dificulta la definición de técnicas reutilizables para distintos dominios y aplicaciones, en CBROnto estamos interesados en identificar y formalizar propuestas genéricas e independientes del dominio que, además, saquen partido del modelo de conocimiento explícito sobre el dominio.

En nuestra aproximación a la tarea de adaptación hemos definido, por un lado, métodos totalmente genéricos –cuyos resultados son limitados– y, por otro lado, métodos que requieren una componente específica del dominio y/o de la aplicación. Estos métodos especializados suponen para el diseñador un esfuerzo adicional de adquisición de conocimiento de adaptación pero obtienen mejores resultados.

Ambas aproximaciones, genérica y especializada, se construyen sobre la idea de considerar el conocimiento de adaptación como una combinación de conocimiento sobre transformaciones generales y sobre búsqueda en memoria. Una de las ventajas de nuestro modelo de representación explícita del conocimiento del dominio es el solapamiento de los roles que juegan los distintos tipos de conocimiento de los sistemas CBR diseñados con COLIBRI. Es decir, todo el conocimiento del que los métodos disponen para razonar, está integrado en una única base de conocimiento. En particular incluye, los casos, el modelo de conocimiento del dominio, la terminología de CBROnto, el conocimiento adicional de similitud y el conocimiento aprendido.

El único requisito que los métodos de adaptación imponen sobre la estructura de los casos es la existencia de solución, que será la componente a adaptar en cada caso. La Figura 4-10 muestra la jerarquía de clasificación conceptual correspondiente a las tareas y métodos de adaptación de CBROnto. Cada método de adaptación se representa como una instancia directa del concepto REUSE_METHOD. Además, sus subconceptos representan los métodos que resuelven las subtareas que se derivan de aplicar los métodos de adaptación. El esquema de tareas y subtareas asociado a los métodos de adaptación está basado en el esquema de [Aamodt&Plaza94] y en el modelo de tareas de adaptación propuesto en [Fuchs99]:

- La primera subtarea de la tarea de adaptación de casos (Copy_Solution_Task) tiene como objetivo hacer una copia de la solución del caso recuperado. Como el caso recuperado no es único, en general, se elegirá el primero (ya que están ordenados por similitud) a menos que tenga un resultado de fallo, en cuyo caso se le preguntará al usuario si quiere utilizarlo.
- La subtarea de adaptación de la solución (Adapt_Solution_Task) se encarga de adaptar la solución de la consulta para que satisfaga los requisitos de su descripción.

En la estructura relacional que definen las instancias canónicas y que hemos representado en la Figura 4-11, el individuo icopy_solution_task es el representante canónico de la tarea Copy_Solution_Task y está ligado a un único método (icopy_solution_method) que realiza una copia sobre el individuo consulta de la componente de solución del caso recuperado. Esta copia (y no la solución original) será modificada en la siguiente tarea.

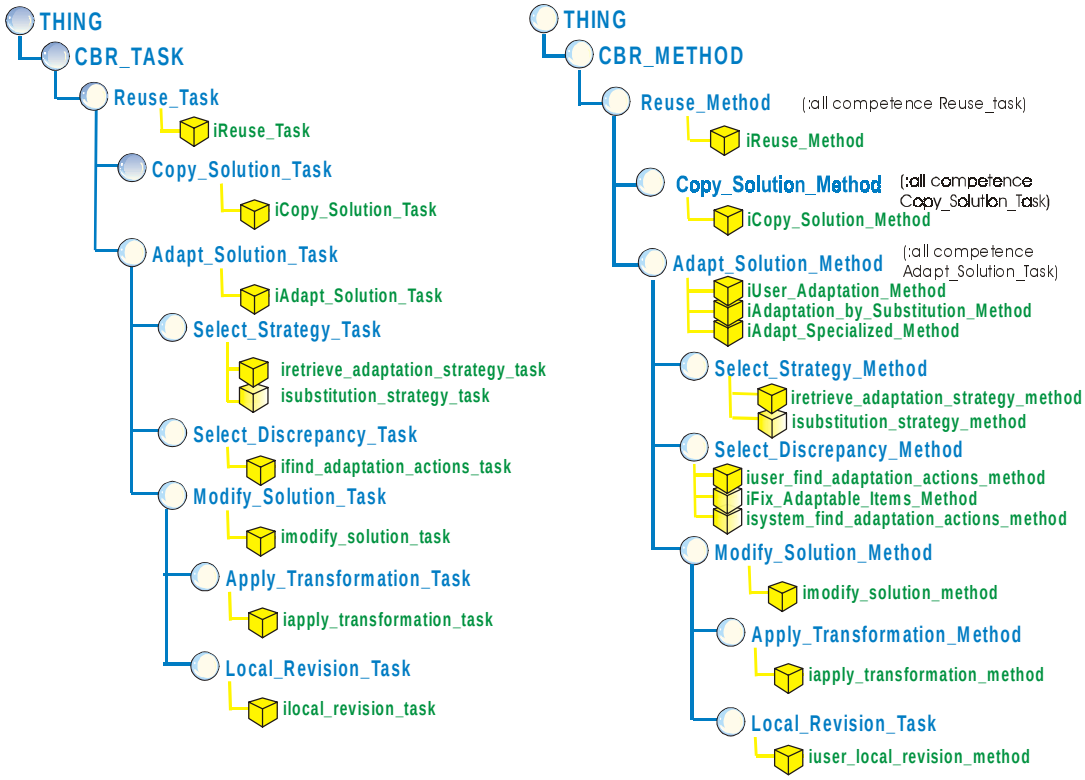


Figura 4-10. *Tareas y métodos de adaptación*

Para la tarea de adaptación o modificación de la solución, cuyo representante canónico es el individuo `iAdapt_Solution_Task`, la inversa de la relación de competencia permite encontrar varios métodos alternativos que se pueden usar para resolverla. Enunciamos aquí los métodos de adaptación cuyo comportamiento se detalla en el Capítulo 5:

- Método de adaptación manual por el usuario. Representado por el individuo `iUser_Adaptation_Method`, es un método de resolución que delega en el usuario la tarea de adaptación de la solución del caso recuperado.
- Método genérico de adaptación basado en sustituciones. Representado por el individuo `iAdapt_by_Substitution_Method` es un método de descomposición que se basa en realizar sustituciones guiadas por el conocimiento del dominio.
- Método de adaptación especializada basada en estrategias (`iadapt_Specialized_Method`) que se basa en recuperar y aplicar una estrategia de adaptación que indica cómo adaptar el caso recuperado, en concreto qué transformaciones utilizar y cómo encontrar los elementos involucrados en las mismas.

Los dos últimos métodos descomponen la tarea de adaptación de la solución en varias subtareas (Figura 4-10):

- Encontrar la estrategia de adaptación a utilizar (`Select_Strategy_Task`).
- Determinar los elementos de la solución sobre los que aplicar las transformaciones especificadas en la estrategia (`Select_Discrepancy_Task`).

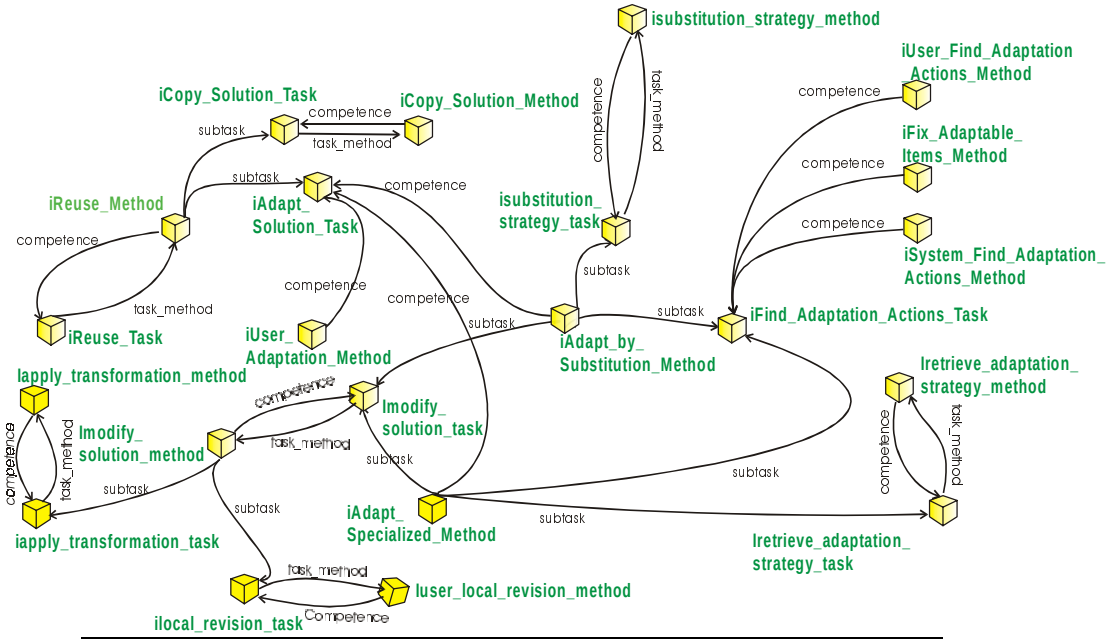


Figura 4-11. *Estructura relacional de las tareas y métodos de adaptación*

- Aplicar las transformaciones a dichos elementos para resolver las discrepancias (`Modify_Solution_Task`).

La resolución de la tarea `Select_Strategy_Task` varía según el método de adaptación de la solución. Mientras el método genérico basado en sustituciones usa una estrategia genérica predefinida –basada en la sustitución de elementos– el método de adaptación especializada usa estrategias específicas definidas por el diseñador de la aplicación. La recuperación de estas estrategias especializadas se basa en los problemas que plantea el caso recuperado en la situación actual.

El objetivo de la tarea `Select_Discrepancy_Task` es determinar qué partes de la solución del caso no son adecuadas respecto a la consulta dada y que, por tanto, deben ser adaptadas. En CBROnto esta tarea está guiada por la representación explícita de las relaciones de dependencia de los elementos de la solución del caso y por la detección de diferencias entre las descripciones de los casos consulta y recuperado.

Cada estrategia de adaptación hace referencia a uno o varios *operadores de transformación* que están predefinidos en CBROnto:

- *Añadir* un elemento, entendiendo como elemento una cadena de relaciones a partir del individuo caso que termina en un individuo (que puede ser terminal o no).
- *Eliminar* un elemento, indicando la posición y el individuo, o eliminar un individuo, es decir, cualquier aparición del individuo en cualquier posición del grafo de representación del caso.
- *Sustituir* un elemento por otro de características "similares" que se adecue también a las características de la consulta.

Por tanto, una vez identificados los elementos de la solución susceptibles de transformación, la tarea `Modify_Solution_Task` se encarga de llevar a cabo los pasos de transformación

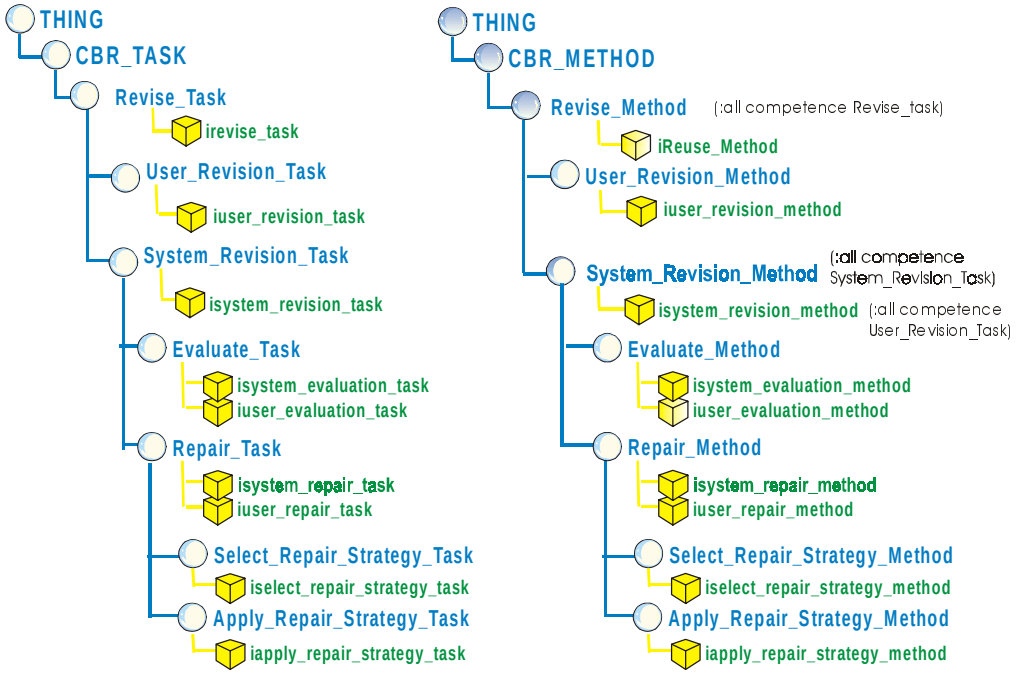


Figura 4-12. *Tareas y métodos de revisión*

representados en la estrategia sobre los elementos discrepantes. La validez de cada paso de transformación puede ser comprobada por el usuario.

Observamos que un método de adaptación que se basa en la sustitución de componentes se comporta especialmente bien para la adaptación de casos con una estructura relacional similar a la especificada en la consulta pero que difiere en los rellenos de las relaciones. La adaptación consiste entonces en sustituir los rellenos del caso recuperado por otros que satisfagan ciertos criterios especificados generalmente también en la consulta. Este método está relacionado con la adaptación en sistemas de diseño en los que el objetivo es completar un diseño dado que está a "medio hacer".

3.3.4 Tarea y métodos de revisión

Una vez que la solución del mejor caso recuperado ha sido adaptada, la tarea de revisión se encarga de evaluar su adecuación a la situación actual.

Una de las opciones ofrecidas en CBROnto, llevada a cabo en la mayoría de los sistemas, consiste en delegar la resolución de la tarea de revisión (REVISE_TASK) en el usuario del sistema, que valida la solución generada.

Para poder añadir métodos de revisión automática se requiere una representación explícita de la tarea que resuelve el sistema, por ejemplo, en términos de los objetivos que satisface un caso correcto. De esta forma dado un caso adaptado seremos capaces de valorar qué objetivos satisface y compararlos con los requisitos de corrección. Como describiremos en el Capítulo 5, la revisión automática es similar al método de adaptación por estrategias especializadas, y se basa en determinar los tipos de fallos y estrategias de reparación asociadas a los mismos.

Como se muestra en la Figura 4-12, en CBROnto existe un único método de revisión iRevise_Method (instancia de REVISE_METHOD) asociado a la tarea iRevise_Task, que descompone la tarea de revisión en dos subtareas: revisión automática (iSystem_Revision_task) y revisión manual (iUser_Revision_task). Las dos subtareas no son obligatorias. Además, para cada una de las dos subtareas anteriores existen sendos métodos que las resuelven: método de revisión automática –iSystem_Revision_method que a su vez genera dos subtareas: evaluación y reparación– y método de revisión manual –iUser_Revision_method.

El método de evaluación automática –iSystem_Evaluation_Method– es un método de resolución que se basa en clasificar el caso adaptado y comparar los conceptos bajo los que está clasificado con la clasificación del caso recuperado. En función de las diferencias en la clasificación se identifican los problemas o fallos que tiene el caso propuesto. Una vez identificados los problemas comienza la resolución de la subtarea de reparación automática –iSystem_Repair_Method– en la que los problemas encontrados sirven de índices para recuperar las estrategias de reparación adecuadas. Realmente el flujo de control define un ciclo entre las tareas de evaluación y reparación, de forma que en cada vuelta se identifica un fallo y se intenta su reparación.

Al igual que el método de adaptación utilizando estrategias, el método de reparación automática descompone la tarea de reparación en dos subtareas: encontrar estrategia –Select_Repair_Strategy_Task (individuo iselect_repair_strategy_task)– y aplicar estrategia –Apply_Repair_Strategy_Task (iapply_repair_strategy_task).

El método que aplica la estrategia devuelve un resultado de éxito o fallo, de forma que si la estrategia falla y no es capaz de reparar el error, dejaremos que lo arregle el usuario en la tarea de revisión manual.

3.3.5 Tarea y métodos de aprendizaje

La última tarea que se resuelve en un ciclo CBR es la tarea de aprendizaje de la experiencia. Como vimos en el Capítulo 2, el CBR es un enfoque de aprendizaje incremental ya que el sistema actualiza su conocimiento en base a nuevas experiencias que pasan así a estar disponibles para abordar futuros problemas. Como se muestra en la Figura 4-13, CBROnto define un método (iRetain_Method) que resuelve la tarea de aprendizaje (iRetain_Task) descomponiéndola en dos subtareas:

- Aprender un nuevo caso que se incorpora a la base de casos (iretain_case_task). El primer problema que se debe tratar es decidir qué casos se aprenden ya que, como vimos en el Capítulo 2, la eficiencia de un sistema CBR se puede degradar cuando el número de casos crece excesivamente y, por lo tanto, se debe evitar incluir casos que no aporten información nueva al sistema.
  El comportamiento del método que resuelve esta subtarea se basa en añadir los casos cuya revisión es positiva a una base de casos temporal. El aprendizaje definitivo no se produce hasta que, periódicamente, un usuario experto deberá decidir la inclusión definitiva de estos casos en la base de casos con la que trabaja el sistema CBR. Esto se lleva a cabo cuando se resuelve la tarea de mantenimiento del sistema (CB_Maintenance_Task). Una ventaja derivada del uso de un sistema de DLs, es que la inclusión de un nuevo caso es sencilla ya que gracias a sus mecanismos de razonamiento, en particular, al reconocimiento de instancias, el sistema es capaz de colocar el nuevo caso en el lugar que le corresponda en la estructura de organización.

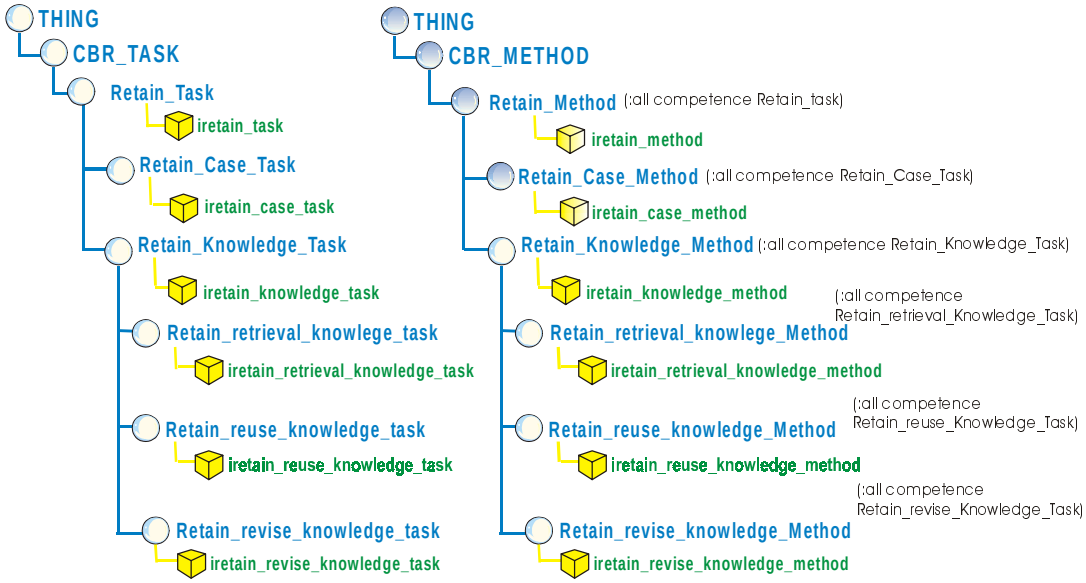


Figura 4-13. *Tarea y métodos de aprendizaje*

- Aprender conocimiento de recuperación, adaptación y revisión (`iretain_knowledge_task`)
  - Añadiendo resultados de éxito o fallo en el caso recuperado.
  - Pesando positiva o negativamente las estrategias de adaptación y reparación utilizadas.

  Esta subtarea sólo tiene sentido si durante el diseño de la aplicación se han definido estrategias específicas de adaptación y revisión de casos. El método –`iretain_knowledge_method`– que resuelve esta subtarea deriva a su vez tres subtareas: `iretain_retrieval_knowlege_task`, `iretain_reuse_knowledge_task` e `iretain_revise_knowledge_task` que se encargan del aprendizaje de los distintos tipos de conocimiento asociados a cada una de las tareas del ciclo CBR.

Aunque muy importante desde el punto de vista del CBR, la tarea de aprendizaje ha sido la que menos atención ha recibido en CBROnto. Aunque hemos incluido algunos mecanismos para aprender cierto conocimiento los métodos de aprendizaje actuales no son autónomos en el sentido de que no deciden qué conocimiento incluir sino que delegan esta responsabilidad en un usuario experto.

4. La terminología CBR de CBROnto

En el apartado anterior hemos introducido CBROnto como una ontología de tareas y métodos CBR y una biblioteca de métodos que instancia dicha ontología y que formaliza métodos genéricos, aplicables en distintos dominios y aplicaciones. La definición de estos métodos genéricos es posible sólo si se pueden asumir ciertos *compromisos* (requisitos de aplicabilidad de los métodos) por ejemplo, respecto a los casos que manejan los métodos, al conocimiento del dominio existente o a la estructura de dicho conocimiento, que determinará los pape-

les que los términos del dominio juegan en los métodos. Recordamos la arquitectura de dos capas (Figura 4-2) en la que la terminología de CBROnto sirve como puente de conexión entre el modelo del dominio de la capa inferior con los métodos de la capa superior. En este apartado describimos la terminología CBR de CBROnto que se utiliza como el vocabulario básico sobre el que se definen los métodos de su biblioteca.

Con el objetivo de estructurar conceptualmente el contenido de CBROnto podríamos distinguir entre varios tipos de definiciones terminológicas:

- Términos de la ontología de tareas y métodos descritos en el apartado anterior, que contiene los términos involucrados en la representación explícita de las estructuras de tareas y métodos.
- Términos de lenguaje de descripción de casos y definición de tipos de casos.
- Términos que representan roles de conocimiento utilizados en los métodos (medida de similitud, objetivo, precondición, acción, etc).
- Otros términos que ayudan a la clasificación, por ejemplo, los distintos tipos de relaciones o atributos, o los términos que se utilizan para organizar las jerarquías de conceptos y relaciones del dominio y de CBROnto.

La división se ha realizado con el objetivo de una presentación más clara de los contenidos de la ontología, pero la frontera entre las facetas anteriores no está claramente delimitada. La ontología está formalizada como una única base de conocimiento en LOOM en la que encontramos 198 conceptos, 98 relaciones y 102 individuos.

El apartado siguiente describe los aspectos relativos la representación de casos, haciendo hincapié en el lenguaje de representación de casos que ofrece CBROnto. En el Apartado 4.2 introducimos el tipo de integración terminológica basada en clasificación que proponemos para conectar el modelo del dominio con los métodos de la biblioteca de CBROnto.

4.1 Representación de los casos

Los casos son las unidades básicas de conocimiento con las que un sistema CBR razona. En un sistema CBR cada caso se describe usando constructores de un cierto lenguaje que permite especificar la estructura, las restricciones y el contenido de los casos. Además, se debe tener en cuenta que el formalismo elegido para representar los casos tiene una influencia directa en los métodos que razonan con estos casos.

Existen dos cuestiones distintas, aunque relacionadas, relativas a la representación de casos. Por un lado, el lenguaje debería permitir definir *tipos de casos* para representar estructuras adecuadas para describir el contenido de los casos. Cada tipo de casos especificará qué descriptores se utilizan, si existe algún descriptor obligatorio (se marcarán utilizando la relación de CBROnto `mandatory-relation`), si queremos diferenciar entre distintas partes, descripción, solución y resultado (usando las relaciones `has-description`, `has-solution` y `has-result`), o no, etc. Por otro lado, el lenguaje se utiliza para definir los casos que guardan la información concreta estructurada según alguno de los tipos.

CBROnto define un lenguaje de descripción de casos que permite representar tipos de casos y casos concretos de esos tipos. Una característica adicional es la inclusión de algunos tipos de casos predefinidos que son adecuados en aplicaciones prototípicas del CBR, como la interpretación, la planificación o el diseño.

En el Capítulo 2 hemos introducido distintas posibilidades para representar casos que están presentes en la literatura de CBR, desde lenguajes muy formales hasta descripciones informales, textos, gráficos, etc. Nuestra aproximación propone utilizar representaciones en

LOOM, formales y estructuradas –vs. planas– con las que los métodos CBR razonan. Además de los constructores del lenguaje LOOM, las representaciones de los casos usan terminología propia de CBR del lenguaje de representación de casos de CBROnto.

Suponer que los casos son unidades monolíticas de formato fijo contradice la flexibilidad de uso del conocimiento representado en los casos. Por tanto, hemos adoptado una aproximación flexible y descomposicional para representar los casos, que permita que los fragmentos de los casos sean utilizados de forma aislada y que múltiples casos sean combinados para generar una solución para un problema.

En el esquema de representación que hemos elegido, cada caso es un individuo, instancia (directa o a través de alguno de los subconceptos) del concepto `CASE` de CBROnto. En las estructuras de los casos pueden aparecer relaciones del dominio o relaciones de CBROnto, que permiten que un caso se relacione con otros individuos que a su vez pueden ser estructurados. Ya que las instancias del concepto `CASE` pueden estar categorizadas de diversas formas, y no podemos dar una definición de las condiciones necesarias y suficientes de pertenencia al concepto, `CASE` es un concepto primitivo que se define por extensión, es decir, por el conjunto de sus instancias. El conjunto de instancias de `CASE` determina la base de casos de la aplicación. Además, `CASE` es la raíz de una jerarquía de conceptos cada uno de los cuales representa un tipo de casos. Cada uno de estos tipos de casos se define según sus propiedades (necesarias y suficientes) por lo que, dependiendo de la definición, el sistema puede reconocer el tipo al que pertenece un cierto caso.

Aunque existen algunos tipos de casos prototípicos predefinidos en CBROnto, durante el diseño de un nuevo sistema CBR se pueden definir nuevos tipos específicos de la aplicación concreta, y ambos se representan en CBROnto como subconceptos de `CASE`. Realmente para distinguir los tipos de casos de otros subconceptos definidos de `CASE`[2] la jerarquía de tipos de casos se organiza bajo el concepto `CASE-TYPE` que es un subconcepto directo de `CASE`. Aunque no es necesario, los tipos de casos específicos de una aplicación concreta pueden definirse en base a alguno de los tipos predefinidos de CBROnto. Para especificar tipos de casos se utiliza una sintaxis similar a la del lenguaje de descripción de casos.

En CBROnto cada *caso* -instancia del concepto `CASE` de manera directa o por pertenencia directa a alguno de sus subconceptos que representan tipos de caso– se relaciona a través de las relaciones `has-description, has-solution y has-result` con individuos que representan, respectivamente, la *descripción*, la *solución* y el *resultado* del caso [Kolodner93]:

- El individuo (*d*) que representa la descripción del caso es una instancia (directa o no) del concepto `CASE-DESCRIPTION` de CBROnto. El significado o semántica del individuo está determinado por los conceptos a los que pertenece, que pueden pertenecer a CBROnto o al modelo del dominio, y por su relación con otros individuos, que pertenecerán típicamente al modelo del dominio.
- El individuo (*s*) que representa la solución del caso es instancia (directa o no) del concepto `CASE-SOLUTION` de CBROnto.
- Los individuos (*ri*) que representan el resultado del caso son instancias (directas o no) del concepto `CASE-RESULT` de CBROnto. Como en las descripciones, la semántica de estos individuos está determinada por los conceptos a los que pertenecen y por su relación con otros individuos.

[2] Como los conceptos `Case_with_Description` o `Case_with_Solution`, que se definen para comprobar la aplicabilidad de los métodos.

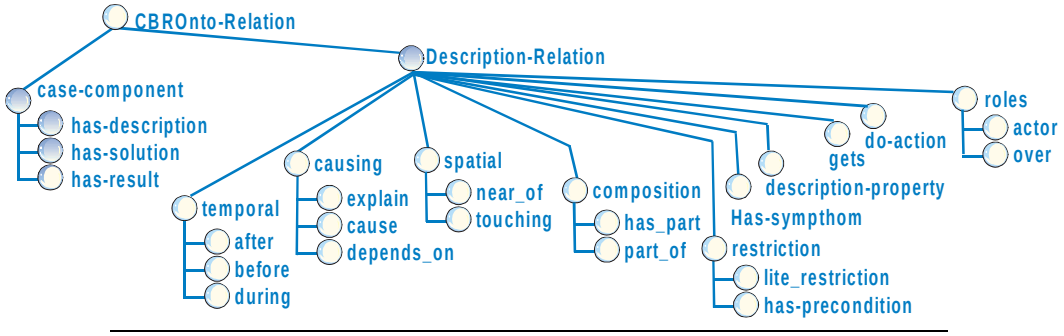


Figura 4-14. *Tipos de relaciones de CBROnto*

Sólo la descripción de los casos es una componente de aparición obligatoria. Por lo demás las estructuras del lenguaje de representación de casos son *libres*, en el sentido de que no se establece obligatoriedad para las componentes. Esta característica permite definir estructuras muy variadas y con distinto nivel de complejidad con las que el sistema es capaz de razonar. La pertenencia a los conceptos de CBROnto puede ser asertada explícitamente o inferida por los mecanismos de razonamiento de LOOM.

Los apartados siguientes describen los tipos de relaciones que aparecen en CBROnto, el lenguaje de descripción de casos que permite especificar tanto los tipos de casos como los casos concretos de esos tipos, y cómo los tipos de casos pueden utilizarse para definir los tipos de consultas que serán válidos durante la interacción con el usuario.

4.1.1 Los tipos de relación

Una de las ventajas de utilizar un sistema de DLs expresivo como LOOM es que nos permite trabajar con jerarquías de relaciones. En CBROnto hemos realizado una clasificación jerárquica de distintos tipos de relaciones que pueden aparecer en un dominio o al describir los casos (Figura 4-14). La representación explícita de estos tipos de relaciones nos ayuda a encontrar patrones relacionales en los casos, así como a conseguir procesos de razonamiento específico basados en un proceso general de clasificación y recorrido de la jerarquía de relaciones.

Cada una de las relaciones de la Figura 4-14 puede utilizarse directamente en la definición de casos o indirectamente a través del mecanismo de integración que se basa en clasificar las relaciones del dominio bajo las relaciones de CBROnto. Es decir, durante la integración podemos clasificar una relación del dominio, por ejemplo, como estructural o como temporal, o como propiedad descriptiva.

Identificar los roles que juegan las relaciones nos permite distinguir por ejemplo la *estructura* o partes que componen el caso; las *propiedades* que lo describen; los *síntomas* que describen un caso; los *objetivos* que consigue el caso; o la secuencia de *pasos de la solución* del caso. Esto permite hacer una distinción semántica de las relaciones del dominio desde el punto de vista del papel que juegan en los métodos CBR. Por ejemplo, permite definir distintas componentes de similitud según el tipo de las relaciones que intervienen: la similitud estructural tendrá en cuenta únicamente las relaciones que definen la estructura de los casos, de forma que la similitud entre dos casos no depende del contenido sino de la estructura de los casos.

4.1.2 El lenguaje de definición de casos

En el Apéndice A se incluye la formalización en LOOM del lenguaje de representación de casos de CBROnto. En este apartado, para proporcionar una visión general de la terminología incluimos la sintaxis EBNF aproximada para dicho lenguaje donde los términos de CBROnto aparecen en cursiva. Aunque incluimos este tipo de sintaxis para facilitar la comprensión a los lectores que no estén familiarizados con la sintaxis LOOM, hay que tener en cuenta que no es completa. Por ejemplo, no incluye la distinción explícita entre los distintos tipos de casos, de descripciones, de soluciones, ni la relación taxonómica entre las componentes que aparece en la ontología. Por otro lado, las definiciones en un sistema de DLs permiten que el sistema realice inferencias sobre las componentes que tampoco es posible representar en el esquema siguiente:

```
<case> ::= <case-description> [<case-solution>]{<case-result>}*
<case-description> ::= has-description <Case-Description instance>
<Case-Description instance> ::= {kind-of-reasoning <reasoning-type>}*
                                  {<relation-attributes>}*
<reasoning-type> ::= <Reasoning-Type instance>
<Reasoning-Type instance> ::= diagnosis | compose | evaluate | explain |
                        design | resolve | search
<relation-attributes> ::= <indexed-domain-relation> <individual> |
                     <mandatory-relation> <individual>
                       <subrelations of Description-Relation>
                        <individual>
<indexed-domain-relation> ::= <subrelations of Domain-relation & Very-Relevant-Relation
> | <subrelations of Domain-relation & Relevant-Relation >
<mandatory-relation> ::= <subrelations of Mandatory-Relation>
<subrelations of Description-Relation>::= <Temporal Relation> |
                              <Causing Relation> | <Spatial Relation> |
                              <Composition Relation> | <Restriction Relation> |
                              <Description Property> | <has-sympthom> |
                              <goals-achivement>
<Temporal Relation> ::= temporal | after | before | during
<Causing Relation> ::= causing | explain | cause | depends_on
<Spatial Relation> ::= spatial | near_of | touching
<Composition Relation> ::= composition | has_part | part_of
<Restriction Relation> ::=restriction | lite-restriction | has-precondition
<Description Property> ::= description-property
<has-sympthom> ::= has-sympthom
<goals-achivement> ::=  gets <Goal instance>
<case-solution> ::= has-solution <Case-Solution instance>
<Case-Solution-instance> ::= <Transformational Solution> | <Derivational Solution>
<Transformational Solution> ::= {<relation-attributes>}*
<Derivational Solution> ::= {<solution-steps>}*
<solution-steps> ::= has-sequence-step <Solution-Step instance>
<Solution-Step-instance> ::=<action> <over> <actor> <sequence-number>
<action>  ::= do-action <Action instance>
```

```
<over>  ::=  over <Domain-Concept instance>
<actor> ::= actor <Domain-Concept instance>
<sequence-number> ::= Number
<case-result> ::= has-result <Case-Result instance>
<Case-Result-instance> ::= <Success-Result instance>|<Failure-Result instance>
<Failure-Result instance> ::= [<failure-solution>][<repaired-solution>]
                          <repair-strategy>
<failure-solution> ::= failure-solution <Failure-Case-Solution instance>
<repaired-solution> ::= repaired-solution <Case-Solution instance>
<repair-strategy> ::= repair-with <Repair-Strategy instance>
```

Nuestro lenguaje de representación de casos se ajusta al esquema general de representación de casos estructurados descrito en el Capítulo 2 (Apartado 4.2.1) si puntualizamos ciertas diferencias:

- Las relaciones binarias son equivalentes a los atributos relacionales.
- Aunque en nuestro esquema de representación de casos los "rellenos" siempre son individuos, se considera que un atributo es simple cuando sus rellenos son individuos sin atributos, es decir, he llegado a un nodo terminal del grafo de representación del objeto. En particular, esto ocurre en los conceptos simples y en los conceptos que representan a los tipos básicos como `Number`, `String` o `Symbol`.
- En nuestro esquema se permiten atributos multivaluados o multivalorados. El relleno de un atributo en un objeto (individuo) es el conjunto de individuos con los que se relaciona a través de ese atributo o relación binaria.
- Existen jerarquías taxonómicas de relaciones. Si una clase de individuos determina la existencia de un atributo relacional para sus instancias, cualquier subrelación suya también es adecuada para las instancias (objetos) de esa clase.
- La terminología de CBROnto ofrece ciertos términos, tanto relaciones como conceptos, con semántica propia y con los que se puede trabajar de un modo específico, por ejemplo, los distintos tipos de relaciones (temporal, estructural, causal, etc) o conceptos como `Goal` o `Action`.
- En el esquema general la estructura de un objeto está totalmente determinada por la clase a la que pertenece. En nuestro esquema, aunque dos individuos sean instancias de la misma clase no tienen porqué tener exactamente la misma estructura, porque un individuo puede incluir atributos específicos no heredados de ninguna clase, y además, la herencia múltiple hace que un individuo pueda heredar atributos de distintas clases.

Una idea importante de nuestro esquema de representación, que ya hemos comentado, es que las representaciones son flexibles es decir, que aunque CBROnto ofrece términos privilegiados con los que construir estructuras específicas para una aplicación concreta no existen demasiadas restricciones sobre las estructuras concretas que se pueden construir. Los términos utilizados determinan los papeles que juegan los términos del dominio en los casos y se utilizarán durante el razonamiento.

Sin embargo sí el importante diseñar estructuras con las que los métodos puedan razonar convenientemente. Para facilitar este proceso CBROnto ofrece algunas estructuras predefinidas en los conceptos que describen los tipos de casos (ver Apartado 4.1.3). Un diseñador puede utilizar estas estructuras, que utilizan las relaciones y conceptos de CBROnto de la

forma requerida por los métodos, y particularizarlas usando el conocimiento del dominio de la aplicación. Otra opción es que el diseñador defina nuevas estructuras para los casos. Sin embargo, para que los métodos puedan razonar con dichos casos, deberá utilizar parte de la terminología de CBROnto (directa o indirectamente a través de la clasificación). Por ejemplo, distinguir qué parte de la estructura se corresponde con la descripción del caso, utilizando la relación `has-description` de CBROnto.

Los siguientes apartados describen la terminología de CBROnto para representar las descripciones, soluciones y resultados de los casos.

4.1.2.1 Representación de las descripciones de los casos

La descripción de un caso representa la situación, el estado del mundo antes de comenzar el razonamiento que representa el caso. Dependiendo del tipo de CBR la descripción del caso puede ser una situación a interpretar o clasificar, un problema a resolver o la descripción de un elemento a diseñar. Para casos con solución, la descripción también puede incluir características que limiten el contexto al que es aplicable la solución del caso y la descripción de los objetivos que satisface la aplicación de la solución del caso.

Para representar la descripción de un caso se utiliza el individuo con el que un caso concreto se relaciona a través de la relación `has-description` (directamente o indirectamente por clasificación de una relación del dominio como especialización suya). La descripción de un caso siempre es instancia (directamente o a través de algún subconcepto) del concepto CASE-DESCRIPTION.

Una de las relaciones que participan en la descripción de un caso es la relación `kind_of_reasoning` de CBROnto que representa el tipo de razonamiento que lleva a cabo el caso: "diagnosticar", "evaluar", "explicar", "diseñar", "resolver", etc. Además, las descripciones pueden incluir diversas características descriptivas del problema o situación que representa el caso. Para ello se utilizan relaciones con otros individuos, típicamente relaciones del modelo del dominio clasificadas (o no) bajo las relaciones de CBROnto. La clasificación de relaciones permite identificar varios tipos de relaciones (temporales, causales, descriptivas, etc) según el papel que juegue en la descripción.

Por ejemplo, el tipo de relación `restriction` permite incluir restricciones de aplicabilidad del caso, que son propiedades que debe cumplir una situación antes de poder aplicar la solución del caso, tanto recomendadas (`lite-restriction`) como obligatorias (`has-precondition`). La relación `gets`, o cualquier especialización suya, relaciona la descripción del caso con los objetivos (instancias del concepto GOALS) que se cumplen después de aplicar la solución del caso.

Este marco de representación también permite definir *tipos de descripciones* cada una de las cuales representa la estructura común de las descripciones de los casos de alguno de los tipos de casos. Un tipo de descripción se representa como un concepto que especialice al concepto CASE-DESCRIPTION de CBROnto. Existen distintos tipos de descripciones predefinidas en función de los tipos de relación que intervienen en ellas: descripción espacial, temporal, de diseño, descriptiva, etc. (las definiciones se pueden consultar en el Apéndice A).

Dependiendo de la complejidad de la aplicación, la estructura o tipo de descripción puede corresponder directamente con un concepto del dominio. Una opción, que se muestra en la Figura 4-15, es que el mismo individuo caso (instancia de CASE) coincida con el individuo descripción (instancia de CASE-DESCRIPTION). En este ejemplo VIAJE es el tipo de descripción de los casos de tipo CASO-VIAJE. También se pueden definir nuevos conceptos específicos que aúnen distintos términos del dominio en una descripción conceptual compleja.

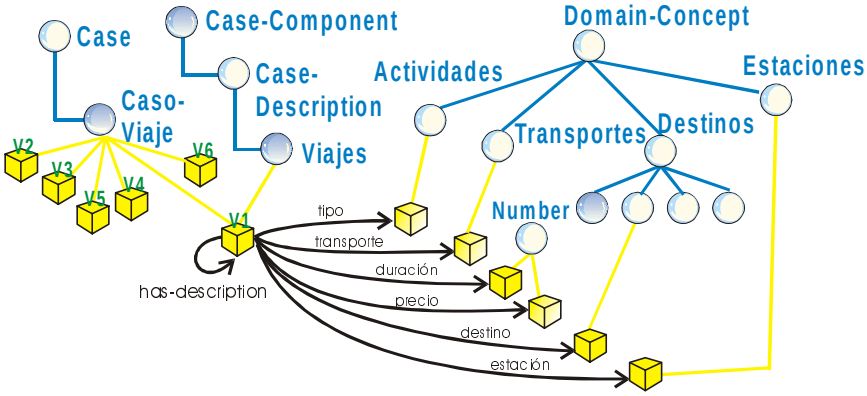


Figura 4-15. *Ejemplo de representación de un caso sencillo*

4.1.2.2 Representación de las soluciones de los casos

La solución de un caso debe representar de algún modo el proceso de razonamiento que soluciona el problema representado en la descripción del caso. Aunque el tipo de soluciones depende en gran medida del tipo de razonamiento, en líneas generales nuestro lenguaje utiliza como piezas constructoras las siguientes componentes:

- En aproximaciones transformacionales al CBR, se almacena la propia solución, que según el tipo de CBR puede consistir en un elemento diseñado, un plan confeccionado, o una interpretación o diagnóstico. En este caso el individuo que representa la solución se describe en términos del dominio, es decir, el sistema lo reconocerá como instancia del concepto CASE-SOLUTION de CBROnto, pero sus características serán asertadas usando los conceptos y relaciones del dominio, que pueden estar clasificadas bajo los términos de CBROnto para indicar el papel que juegan (temporal, estructural, etc).
- En aproximaciones CBR derivacionales, se almacena la secuencia de pasos de razonamiento que se han utilizado para resolver el problema. Un individuo solución puede estar descrito por varios pasos de una secuencia de acciones. La relación `has-sequence-step` relaciona la solución con un individuo que representa un paso de razonamiento consistente en una secuencia de acciones (con las que se relaciona a través de la relación `do-action`) aplicadas sobre un individuo (con el que se relaciona a través de la relación `over`) y realizada por un cierto actor (con el que se relaciona a través de la relación `actor`).

4.1.2.3 Representación de los resultados de los casos

El resultado de un caso representa información sobre lo que ha ocurrido al aplicar el caso en distintas situaciones. El objetivo es que la información representada permita predecir si se debe aplicar una solución o no, anticipar y explicar fallos y obtener soluciones mejores que no cometen fallos previos. Un caso puede tener distintos resultados que se representan mediante individuos (instancias de CASE-RESULT) que se describen en términos del dominio.

Cada individuo resultado se clasificará bajo uno de los conceptos que indican si el resultado es de éxito o de fracaso: FAILURE-RESULT o SUCCESS-RESULT. Un resultado de fallo puede incluir características adicionales que expliquen el fallo, por ejemplo enlaces con las solu-

ciones fallidas, enlaces con alguna estrategia de reparación de la solución del caso, y enlaces al resultado de la reparación del caso.

4.1.3 Tipos de casos predefinidos

Como ya hemos descrito, en CBROnto existen conceptos predefinidos que especializan al concepto `CASE` –a través de su subconcepto `CASE-TYPE`- y que representan ciertos tipos de casos prototípicos. Aunque estos tipos de casos se pueden utilizar directamente para definir los casos como instancias directas suyas, normalmente se utilizarán como conceptos base de los que heredar la estructura cuando el diseñador defina los tipos de casos específicos para una aplicación concreta. De esta forma, y ya que la pertenencia a un tipo de caso no inhibe la pertenencia a otro, podemos aprovechar la herencia múltiple del sistema de DL para que un tipo de caso definido por el usuario incluya características de varios de los tipos predefinidos. Esto resulta útil ya que los tipos de casos definidos por el usuario pueden heredar el conocimiento adicional relativo a este tipo de casos. Este conocimiento anotado en los tipos de casos permite que CBROnto pueda *aconsejar* a un diseñador de aplicaciones, por ejemplo, qué métodos o qué medida de similitud usar en función de las características de los casos.

Aunque ya se ha comentado, es importante incidir en la idea de que en una misma base de casos pueden coexistir distintos tipos de casos cada uno con sus particularidades. En cierto modo, las instancias de cada uno de los tipos de casos de los niveles inferiores de la jerarquía se pueden considerar como una (sub)base de casos distinta sobre la que se pueden aplicar métodos específicos y distintos de los utilizados para otros tipos de casos. Normalmente nos referiremos a una única base de casos –definida por todas las instancias de `CASE`- en la que pueden coexistir casos de distintos tipos ya que, aunque en un cierto instante estemos considerando un único tipo de casos, el resto de tipos y casos representan conocimiento adicional sobre el dominio, que también se aprovecha en los métodos CBR.

El reconocimiento de instancias del sistema de DL, y la definición adecuada de los conceptos, permite que los casos se clasifiquen automáticamente en los distintos tipos (por reconocimiento de instancias) según su estructura. En particular los tipos de casos se definen según el tipo de los componentes que los describen, por ejemplo, se reconoce como un *caso con descripción* (instancia del concepto `Case_with_Description`) a cualquier individuo, instancia de `Case`, que tenga un atributo `has-description`. De forma similar se reconocen los casos con solución (`Case_with_Solution`) y los casos con resultado (`Case_with_Result`). Además, los tipos de relaciones permiten una clasificación adicional de los casos en función de los tipos de relaciones que intervienen en sus componentes.

Por ejemplo la definición conceptual siguiente:

```
(defconcept Case_with_Goals is
 (:and Case (:some has-description Description_with_Goals)))
(defconcept Description_with_Goals :is
 (:and Case-Description (:at-least 1 gets) (:all gets Goal)))
```

Permitirá reconocer como instancias suyas a los casos que están descritos a través de los objetivos que satisface. En este tipo de casos resulta adecuada la aplicación de ciertos métodos que razonan sobre los objetivos conseguidos por los casos (descritos en el Capítulo 5).

En el Apéndice A se pueden encontrar las definiciones correspondientes a los tipos de caso (subconceptos definidos de `CASE`) siguientes que se basan en los tipos de relaciones que aparecen en sus componentes:

- Según el tipo de descripción podemos distinguir entre varios tipos de casos. Además del tipo `Case_with_Goals`, distinguimos también entre `Case_with_Restrictions`, que son los casos cuya descripción incluye alguna restricción (`has-precondition` o `lite-restriction`), `Case_with_Compound_Description`, que son los casos cuya descripción tiene alguna relación de composición (especialización de `composition`), `Case_with_Temporal_Description` que reconoce los casos cuya descripción tiene alguna relación `temporal`, y de forma análoga `Case_with_Spatial_Description` y `Case_with_Descriptive_Description`, que reconoce los casos cuyas descripciones incluyen alguna relación espacial (`spatial`) o con propiedades descriptivas (`description-property`), respectivamente.
- Igualmente, según el tipo de solución distinguimos entre `Case_with_Compound_Solution`, `Case_with_Temporal_Solution`, `Case_with_ Spatial_Solution`, `Case_with_Descriptive_Solution`, `Case_with_Derivational_Solution`, `Case_with_Transformational_Solution`.

Las definiciones anteriores se utilizan para identificar tipos de casos adicionales en función de su estructura general –subconceptos de `CASE-TYPE`. Por ejemplo:

- Casos descriptivos (`Descriptive_Case`) que son los casos que se utilizan para describir algo de forma estática en función de sus características descriptivas observables. Son casos con una descripción de tipo `Case_with_Descriptive_Description`. Este tipo de casos es el más simple y no impone restricciones adicionales a la estructura o al contenido de los casos, que pueden tener solución o resultado o no. Este tipo de casos es característico de los sistemas CBR de diagnóstico, en los que se suelen utilizar casos con descripción y cuya solución es simple o inexistente.
- Casos estructurales o compuestos (`Compound_Case`) que son los casos con una descripción estructural (`Case_with_Compound_Description`).
- Casos de diseño (`Design_Case`) que son los casos en los que la descripción y la solución coinciden, es decir, son el mismo individuo. El resultado es opcional. Como especialización de los casos de diseño distinguimos entre:
  - Casos de diseño temporal (`Temporal_Design_Case`) que son casos de diseño con una descripción (y solución) temporal (`Case_with_Temporal_Description` y `Case_with_Temporal_Solution`).
  - Casos de diseño estructural (`Compound_Design_Case`) que son casos de diseño con una descripción (y solución) con estructura `Case_With_Compound_Description` y `Case_With_Compound_Solution`.
  - Casos de diseño arquitectónico (`Architectural_Design_Case`) que son casos con una descripción (y solución) de diseño arquitectónico. La estructura y terminología utilizada en este tipo de casos se basa en el trabajo de [Griffith&Domeshek96] donde se han identificado varios tipos de características para describir espacios. Algunas de estas características se usan para describir los espacios en sí mismos y otras para describir las relaciones entre los espacios. La primera característica describe la *organización de los espacios* y representa el tipo de relación entre los espacios, la fuerza de las relaciones entre ellos (que medirá la importancia de las relaciones anteriores), la distancia y la orientación de los espacios respecto al sitio. El resto de características describen los espacios individuales. Por ejemplo, los *roles* determinan quién usa los espacios y para qué, existen los roles *primario* y *secundario* para distinguir quién usa el espacio frecuentemente y quién de vez en cuando, y un rol *pro-*

piedad que representa objetos no animados asociados con los espacios. La característica *exterior* describe la distribución exterior incluyendo materiales y forma tridimensional. Por último, la característica *interior* describe la función, materiales y características del espacio interior como tamaño, luz y confort térmico. La formalización de esta terminología en LOOM se puede consultar en el Apéndice B.

Todos los casos de diseño comparten los métodos recomendados aunque varían, por ejemplo, en el tipo de similitud que se utilice, principalmente en el tipo de relaciones que se usan. En un sistema CBR de diseño, en los que la descripción y la solución coinciden, las consultas típicas consistirán en descripciones parciales de un diseño para que sea completado por el sistema. Pueden existir ciertas restricciones entre las componentes, dadas por el usuario, por ejemplo, restricciones de estilo o de funcionalidad. De esta forma, además de estructura también se puede describir funcionalidad cuando las colaboraciones entre las distintas componentes de los casos se lleven a cabo para lograr una cierta funcionalidad global.

- Casos de planificación (`Planning_Case`) cuya descripción incluye precondiciones y objetivos (`Case_with_Restrictions` y `Case_with_Goals`) es decir, describe la situación del mundo necesaria antes de llevar a cabo las acciones de la solución, y los objetivos que se cumplen después de aplicarlas. La solución de los casos de planificación está descrita por una secuencia de acciones (`Case_with_Derivational_Solution`). Para este tipo de casos se asocia un ciclo CBR en el que la tarea de recuperación está ligada al método de recuperación por objetivos/precondiciones y la adaptación repite la traza de acciones almacenada en la solución, adecuando los pasos a los objetivos de la consulta.

  Esta aproximación está claramente influida por la literatura sobre sistemas de planificación [Veloso *et al.* 96] [Muñoz *et al.* 99]. La ventaja subyacente es que el tipo de problemas que se pueden resolver está claramente definido y existen criterios para determinar cuándo una solución es correcta. La desventaja es la limitación en la flexibilidad porque sólo se pueden resolver problemas que puedan ser representados por combinación de los objetivos predefinidos (como en el sistema PARIS [Bergmann&Wilke96]).

Esta estructuración de los casos en tipos es flexible, los tipos de casos no son disjuntos y pueden estar interrelacionados. Es decir, se permite que un caso de un tipo sea una parte de otro caso de otro tipo. Puede ocurrir que un caso se clasifique como instancia de varios de estos conceptos, es decir, que pertenezca a distintos tipos de casos. Esto supondrá distintas recomendaciones, de las que el diseñador podrá elegir las que considere adecuadas. Además, como recomendaciones que son, el diseñador puede no elegir ninguna o variar alguna de ellas según su conveniencia.

4.1.4 Tipos de consultas

Un diseñador puede utilizar la terminología de CBROnto para definir cuál es la estructura de las consultas que se permiten en el sistema. La definición de tipos de consultas resulta útil para definir y limitar la estructura de las consultas que se permiten durante el uso de la aplicación diseñada, aunque limita la flexibilidad de uso del sistema.

Para cada tipo de consultas el diseñador define un subconcepto del concepto `Query-Type` de CBROnto con las restricciones adecuadas. Cualquier consulta formulada al sistema debe ser una instancia de alguno de los conceptos más específicos de esta jerarquía y, por tanto,

mantener sus restricciones. En particular, si no se define ninguna especialización cualquier consulta debe ser instancia del concepto `Query-Type` (que no incluye restricciones).

La clasificación de términos de las DLs también es el mecanismo en el que se basa la definición de tipos de consultas. En concreto, resulta útil definir los tipos de consultas como especializaciones (subconceptos) de alguno de los tipos de casos. De esta forma heredan sus características y su estructura, permitiendo también la inclusión de características propias.

El usuario final que interactúa con el sistema diseñado podrá elegir entre alguno de los tipos de consultas existentes (conceptos más específicos de la jerarquía de consultas) y el sistema le ayudará a formular un individuo que satisfaga sus restricciones. La otra posibilidad ofrecida por COLIBRI es permitir la formulación de una consulta de forma libre y dejar que el sistema, usando el mecanismo de reconocimiento de instancias, compruebe la validez de la misma respecto a alguno de los tipos de consulta establecidos durante el diseño del sistema.

4.2 Uso de la clasificación de DLs para integrar el conocimiento del dominio con CBROnto

Como hemos descrito, nuestra arquitectura para diseñar aplicaciones CBR se basa en conectar, mediante clasificación, la base de conocimiento del dominio con la terminología utilizada por los métodos genéricos reutilizables, para que estos puedan ser aplicados. En UPML [Fensel *et al.* 98a] estas conexiones se corresponden con los llamados *puentes*. En concreto en nuestra arquitectura las conexiones son los puentes métodos-dominio de UPML.

El uso de estos puentes entre la base de conocimiento y los PSMs definidos de forma independiente del dominio, es uno de los pilares de nuestra aproximación. La base formal de estos puentes (o adaptadores) es el mecanismo de clasificación semántica de las DLs y el uso de una terminología específica de CBR que se usa de forma consensuada entre los métodos y el dominio. Aunque, en general, el proceso que conecta el dominio con los métodos es uno de los pasos más costosos en la reutilización de bibliotecas de métodos [Park *et al.* 98], en nuestra aproximación este proceso se simplifica porque aprovecha la clasificación de las DLs y la terminología especializada sobre CBR incluida en CBROnto. Durante la fase de integración de conocimiento, el diseñador del sistema debe relacionar manualmente algunos términos del dominio – los de la parte superior de la taxonomía de conocimiento del dominio– con los términos CBR. Esta acción permite *explicar* el dominio desde la perspectiva del CBR y sirve como puente para conectar los PSMs con el conocimiento del dominio.

CBROnto es una ontología genérica y los términos de una ontología sobre un dominio se definen como especializaciones de sus términos. En los sistemas de DLs parte de la semántica de un término la proporcionan los términos por encima de él en la jerarquía de subsunción, es decir, los términos de los que hereda información. En este sentido la clasificación de un término del dominio como especialización de un término de CBROnto proporciona la definición del término del dominio desde el punto de vista del CBR. En concreto se amplía la semántica del término para que pueda utilizarse en una aplicación CBR.

Por ejemplo, el diseñador puede clasificar las relaciones del dominio bajo alguna de las relaciones que determine su tipo (Apartado 4.1.1) temporal, espacial, descriptiva, estructural, etc.; o bajo alguna de las relaciones del lenguaje de representación de casos si definen, por ejemplo, alguna de las componentes de los casos (`has-description`, `has-result` o `has-solution`). También en la jerarquía conceptual se pueden clasificar los términos que representan objetivos (`Goal`), propiedades (`Property`), acciones (`Action`) o componentes de los casos (`Case-Description`, `Case-Solution`, `Case-Result`) entre otras.

Para relacionar nuestra aproximación basada en clasificación con los tipos de *mappings* descritos en el Capítulo 3 (Apartado 4.2.2), diremos que podría ser considerada como una aproximación mixta que cae entre el tipo de *mapping implícito* y el *declarativo*. En relación con el tipo implícito nuestra aproximación comparte su simplicidad conceptual. En cierto modo, se basa en especializar la terminología CBR usada por los PSMs, o generalizar las definiciones del dominio ya que usamos clasificación, para hacer que las definiciones de los términos del dominio satisfagan los requisitos de los métodos, en particular, el vocabulario utilizado por los mismos. En nuestra consideración dicotómica de modelo de conocimiento y método de resolución, adaptamos el modelo de conocimiento para que satisfaga las necesidades del método, y no el método para que haga referencia a los términos del modelo del dominio.

Respecto a la relación con los *mappings declarativos* compartimos la naturaleza descriptiva de las conversiones entre entidades de las componentes. Nuestras relaciones de *mapping* también son especificaciones explícitas de la conversión entre los términos del dominio y los del método, lo que permite mayor claridad en las descripciones del diseñador. La diferencia fundamental es que en nuestro caso no se definen a priori distintas conversiones que pueden llevarse a cabo para traducir objetos entre el modelo de conocimiento y el método, ni utilizamos un intérprete adicional para analizar las declaraciones de las relaciones. En nuestro sistema podemos considerar, en cierto modo, que el clasificador automático del sistema de DL funciona como intérprete, ya que sólo se clasifican manualmente los términos del nivel superior, y los mecanismos de clasificación semántica y herencia, proporcionan las entradas al método en tiempo de ejecución. Con este punto de vista se comparte la ventaja de que la implementación del intérprete es genérica y reutilizable para otras aplicaciones, ya que nos basamos en mecanismos totalmente genéricos.

4.2.1 Ejemplo de integración de conocimiento

Debido a su simplicidad, el ejemplo de la agencia de viajes no ofrece muchas alternativas en cuanto a la integración del conocimiento. Las relaciones del dominio –tipo, transporte, duración, estación, precio, destino– sería apropiado clasificarlas bajo la relación `description_property`, para indicar que comparten el papel de ser propiedades descriptivas.

Un ejemplo más interesante se muestra en la Figura 4-16. Disponemos de terminología genérica (ontológica) relativa a la representación de bloques físicos y su situación espacial dentro de cajas o contenedores. Aprovechando esta terminología, queremos diseñar una aplicación CBR sencilla en la que los casos representan secuencias de planificación de acciones para colocar las piezas de un almacén en ciertos contenedores. Sabemos que en el almacén hay varias piezas o bloques, a las que llamaremos A, B, C, ... y dos tipos de contenedores que varían en la capacidad, y a los que llamaremos Box1 y Box2. Este conocimiento es específico de la aplicación concreta por lo que, durante la fase de modelado del dominio, habrá que identificar y formalizar las definiciones conceptuales correspondientes haciendo uso de la terminología disponible. El mecanismo de clasificación ayuda en la definición de los nuevos conceptos por especialización y herencia de características. Mediante los individuos se representan las instancias o usos concretos de las piezas y los contenedores en los casos.

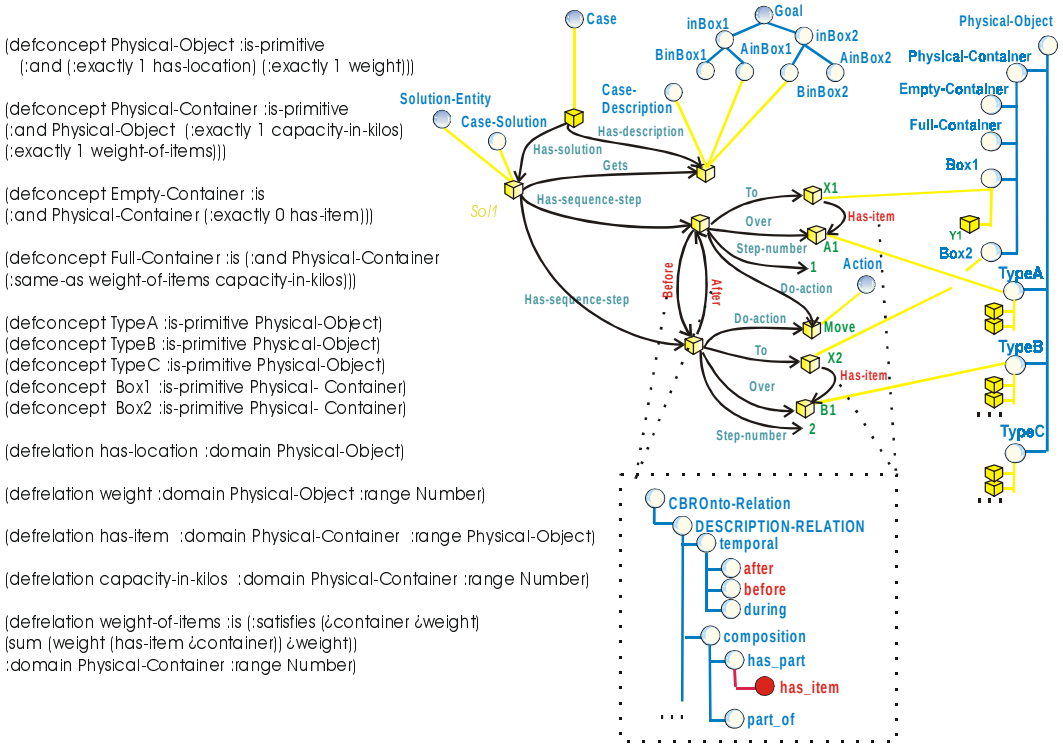


Figura 4-16. *Ejemplo de integración por clasificación*

En la descripción de los casos se indicará la situación inicial de las piezas y la descripción de la situación que quiero obtener y en la solución se indicará la secuencia de movimientos de piezas hasta llegar a la situación pedida. Suponemos una representación de casos basada en el tipo de casos de planificación y, como parte del modelo del dominio, añadimos objetivos básicos, que se definen como subconceptos de `Goal`. Por ejemplo, el objetivo AinBox1 será satisfecho por una situación en la que una pieza del tipo A esté en el contenedor Box1. Los conceptos se definen de tal forma que el sistema reconoce la pertenencia a los mismos según los movimientos realizados en la solución del caso. La inclusión de estos objetivos es opcional y depende de los métodos que se utilicen para razonar con el conocimiento. El tipo de casos de planificación, y los propios métodos a través de los requisitos, sugieren este tipo de adquisición de conocimiento del dominio. Además el individuo que representa la acción de *mover* una pieza será reconocido automáticamente como una instancia del concepto `Action` de CBROnto, por la posición que ocupa en la estructura de representación del caso, que determina el papel que juega.

Otro aspecto importante, que queremos ilustrar con este ejemplo, es la integración o clasificación de las relaciones del dominio bajo las relaciones de CBROnto. Por ejemplo, los métodos "saben" cómo razonar con la relación de CBROnto `has-part` con la semántica adecuada (*contenedor-contenido*) pero no conocen la relación del dominio `has-item` (con el mismo significado). Por esto, el diseñador realizará un paso de integración manual en el que la relación `has-item` se clasifica bajo `has-part` (por lo que se interpreta como relación estructural). Aunque no aparecen en la figura, el resto de relaciones del dominio, como `weight`, `has-location`, `weight-of-items` o `capacity-in-kilos` podrían ser clasificadas como relaciones descriptivas, es decir, bajo la relación `description-property`.

Este tipo de integración se realiza tanto en los conceptos como en las relaciones aunque sólo en los primeros niveles de las jerarquías ya que aprovechamos la clasificación automática y la herencia para propagar la integración a los niveles inferiores.

En el caso de las secuencias de movimientos no es necesaria la integración, ya que en el dominio no se han definido relaciones temporales y CBROnto automáticamente infiere las relaciones `before` y `after` entre los distintos pasos de secuencia de la solución (utilizando el valor del atributo `step-number` de cada paso de secuencia). El proceso de integración sería necesario si en el dominio existiera, por ejemplo, una relación llamada "*previous*". Ésta tendría que ser clasificada bajo la relación `before` para posibilitar el razonamiento.

5. La estructura de la base de casos

Una vez que se ha decidido la estructura y el contenido de los casos como entidades individuales, queda tratar el tema de la organización del conjunto o base de casos de forma que se facilite el acceso a los mismos. Como hemos descrito en el Capítulo 2, éste es el problema de la indexación de la base de casos que conlleva la asignación de índices en los casos y la organización de dichos índices en alguna estructura que permita que los casos similares se recuperen eficientemente.

Existe una relación muy estrecha entre la estructura de organización de la base de casos y los métodos de recuperación definidos sobre ella. Para una cierta medida de similitud habrá que considerar cuál es la organización de la base de casos más adecuada y que permita definir sobre ella un algoritmo de recuperación eficiente que obtenga resultados precisos. En una aproximación como la nuestra, se supone que la asignación de índices en los casos se realiza en un proceso previo de análisis del dominio de aplicación para determinar las características de los casos que se relacionan con el modo en el que se usarán los mismos. Una idea subyacente a nuestro enfoque es la posibilidad del uso de índices con estructuras complejas que facilitan la indexación de casos con estructuras complejas [Martin89].

De las opciones que hemos considerado, la primera supone tener en cuenta *todas* las características que describen a los casos que están inmersos en una red taxonómica de conocimiento general sobre el dominio. Aunque simple, esta opción es adecuada en un esquema de representación formal como el que planteamos y supone que no se seleccionan características o índices específicos sino que se utilizará todo el conocimiento ofrecido por los casos ya que el proceso de análisis del dominio y de los casos se ha realizado previamente. Esta opción permite mayor libertad al formular las consultas.

Una segunda opción consiste en definir como índices un subconjunto de la terminología que participa en la representación de los casos. El conjunto de términos (conceptos y relaciones) índices estará fijado por el diseñador del sistema CBR. CBROnto incluye términos, conceptos y relaciones, para *marcar* cuáles de las características que describen el problema abordado por un caso van a actuar como índices para los casos, es decir, como características predictivas de la utilidad de los casos y a través de las cuáles va a ser posible localizar los casos relevantes para una consulta dada. El diseñador puede utilizar los términos del modelo del dominio –conceptos y relaciones- y marcarlos con la terminología de CBROnto que permite variar su relevancia, o definir nuevos *conceptos índices* que permiten organizar los casos y que completarán el modelo del dominio inicial permitiendo una clasificación más adecuada de los individuos para los objetivos de la aplicación.

El Apartado 5.1 describe algunos aspectos relativos a la organización de los casos. El Apartado 5.2 revisa los términos de CBROnto que pueden ser utilizados para variar la rele-

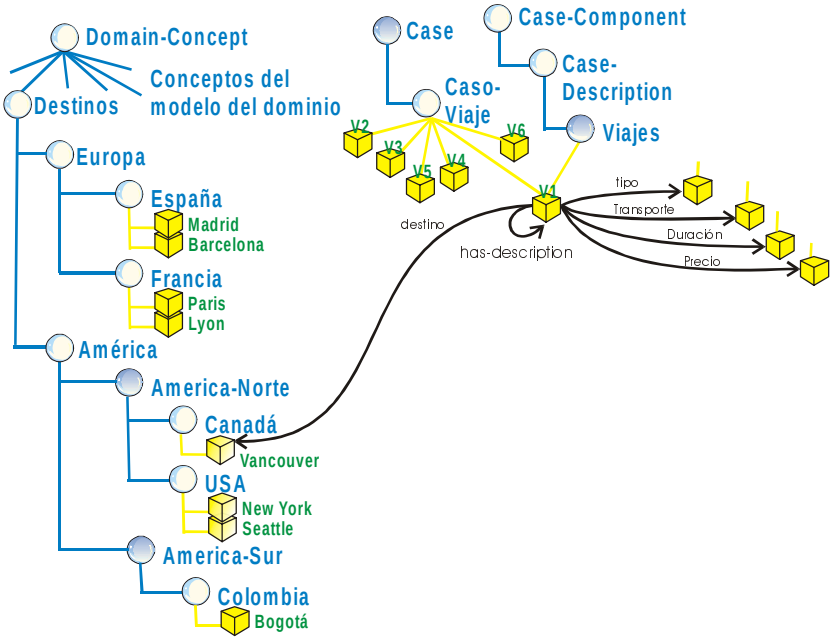


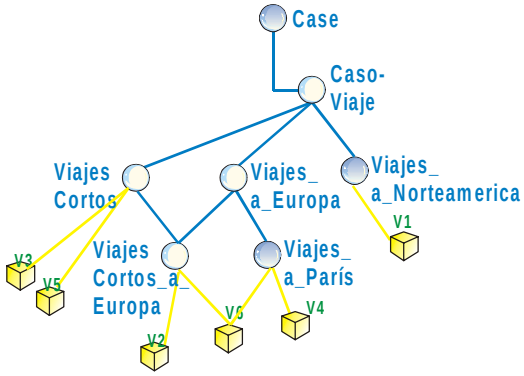
Figura 4-17. *Organización de los casos en el modelo del dominio*

vancia, es decir, para marcar el papel indexador, de los términos del dominio. Por último, el Apartado 5.3 describe el Análisis Formal de Conceptos, una técnica inductiva que hemos utilizado para extraer conocimiento sobre la estructura de un conjunto de casos.

5.1 Organización de los casos

Una vez identificados los índices de los casos, éstos pueden organizarse de manera que el proceso de localización de casos similares se optimice, aunque esta organización no es obligatoria. Las organizaciones clásicas utilizan estructuras precomputadas como árboles de índices o de decisión [Althoff *et al.* 95] o redes discriminantes [Kolodner93]. Aunque sería útil hacer un estudio al respecto, en esta tesis no hemos abordado las estructuras clásicas de organización de casos, sino que hemos aprovechado el hecho de que disponemos de un modelo explícito de conocimiento general sobre el dominio y casos definidos usando terminología de dicho modelo. Esto hace que la propia taxonomía de conocimiento del dominio que pueda considerarse una estructura dentro de la cuál se organizan los casos de forma natural según los términos que los describen.

Aunque esta alternativa es adecuada, hay que tener en cuenta que, obviamente, esta estructura no está optimizada para un método de recuperación concreto, ni es una estructura de índices para explicitar una medida de similitud concreta [Porter89]. Además, aunque la estructura del dominio en sí misma organiza a los individuos que describen a los casos, recursivamente en el grafo de representación, no podemos suponer que organiza *a los propios individuos que representan a los casos*, es decir, a las instancias de `CASE` (ver Figura 4-17). Como ya hemos comentado, el diseñador puede completar el modelo del dominio inicial definiendo nuevos conceptos índices que organizarán la base de casos, es decir, que clasifican las instancias de `CASE` según los objetivos de la aplicación. En la Figura 4-17 los individuos V1, V2, V3, V4, V5 y V6 son las instancias de `CASE` que componen la base de casos. Sin embargo se observa que, aunque la estructura del dominio organiza a los individuos que describen a los




```
(defconcept Viajes_a_Europa :is
          (:and Viajes (:the destino Europa)))
(defconcept Viajes_a_Norteamerica :is
          (:and Viajes (:the destino America-Norte)))
(defconcept Viajes_a_Paris :is
          (:and Viajes (:filled-by destino Paris)))
(defconcept ViajesCortos :is
          (:and Viajes (< duración 7))
(defconcept ViajesCortos_a_Europa :is
          (:And ViajesCortos Viajes_a_Europa)
```

Figura 4-18. *Conceptos de indexación definidos*

casos, por ejemplo, al individuo Vancouver que representa el destino del viaje V1, no organiza directamente a los individuos Vi que representan los casos en sí mismos. Los individuos Vi son instancias de CASE pero sobre ellas no existe ninguna organización. El diseñador puede definir conceptos índices para organizar el conjunto de casos, y aprovechar el reconocimiento de instancias del sistema de DL para que cada individuo caso se clasifique en torno a sus índices (ya que estos son conceptos definidos –ver Capítulo 3). En el ejemplo el diseñador podría definir manualmente conceptos índices que involucren distintas características y utilizando distintos niveles de abstracción como los que se muestran en la Figura 4-18.

Para solventar el posible problema de pérdida de generalidad, se pueden definir conjuntos de índices distintos que ayuden a encontrar el caso en contextos distintos. De esta forma, un mismo caso puede ser indexado por varios conjuntos de índices que se relacionan con distintas alternativas de uso de los casos. Durante un proceso de recuperación el usuario final podrá indicar cuál es el contexto actual de uso del caso eligiendo entre los que se han identificado en la fase de diseño. Esto está relacionado, por ejemplo, con la definición de *perspectivas* del lenguaje Noos [Arcos97] que son expresadas mediante términos de características. Las perspectivas se interpretan como patrones sintácticos que permiten la construcción de descripciones parciales de un problema actual que incluyen solo ciertos aspectos relevantes.

Aunque la definición manual de estos conceptos índices para organizar los casos enriquece el conocimiento del dominio, puede resultar un proceso engorroso para el diseñador de la aplicación, por lo que se puede optar por el uso de la estructura inicial y métodos de recuperación que no requieran estructuras de organización profundas de la base de casos. La otra opción que hemos utilizado consiste en procesar las descripciones de los casos para extraer información de modo inductivo. Aunque en la literatura existen muchas aproximaciones que tienen cabida en nuestro esquema, únicamente hemos estudiado el Análisis Formal de Conceptos (AFC). Esta técnica se describe en el Apartado 5.3 y se aplica para extraer patrones de coaparición de características que permiten agrupar y organizar los casos.

5.2 Términos de indexación de CBROnto

CBROnto incluye ciertos conceptos y relaciones utilizados que permiten *marcar* cuáles de las características predicen la utilidad de los casos. El uso de esta terminología se basa en clasificar los términos elegidos como índices respecto a los términos de indexación de CBROnto. Este proceso se lleva a cabo como parte de la fase de integración del conocimiento del dominio con la terminología CBR, y permite aprovechar la clasificación jerárquica de los términos para clasificar únicamente los conceptos más genéricos de las taxonomías.

En CBROnto suponemos que la elección de las características que predicen la utilidad de los casos es un proceso externo que llevará a cabo el diseñador de un sistema CBR. Una vez elegidas, estas características pueden ser clasificadas bajo los términos de indexación de CBROnto. Realmente el uso de términos de indexación también sería adecuado si se usa alguna técnica automática de selección de índices basada, por ejemplo, en el análisis inductivo de la base de casos que permita predecir la utilidad de las distintas características descriptivas de los casos.

Podemos indexar respecto de alguna de las relaciones usadas para representar casos aprovechando la clasificación de relaciones. Además de las relaciones de CBROnto que indexan las relaciones del dominio según su significado, existen también tres relaciones de indexación generales `very-relevant-relation, relevant-relation, informative-relation` que se utilizan para marcar el nivel de importancia de una relación de cualquier tipo. Cuando se mide la relevancia de un caso respecto a una cierta situación tendrán más peso las relaciones del tipo con relevancia mayor y no se tendrán en cuenta las relaciones informativas.

En el siguiente apartado se describe el lenguaje que define CBROnto para indicar qué tipo de relaciones (atributos) se usarán como índices, es decir, en base a qué tipo de atributos se establece la similitud entre los casos.

5.2.1 Tipos de índices en CBROnto

En CBROnto los *tipos de índice* definen distintas formas de medir la similitud entre individuos en función de las relaciones que contribuyen a la misma. Un tipo de índice `it` se representa como una instancia de `IndexType` que estará ligada con un conjunto de restricciones terminológicas. Cada restricción indica una relación y una posible restricción del rango:

```
<index-type-spec> ::= <Domain-Concept instance> index-type
                                {<IndexType-instance>}*
<IndexType-instance >::= <restriction-desc>
<restriction-desc> ::= index-restrictions {<restriction>}+
<restriction> ::= (<relación> [,<concepto>])+
```

Cuando un método de valoración de la similitud entre dos individuos i1 y i2, aplica un tipo de índice significa que de las relaciones que describen a i1 e i2 sólo tendrá en cuenta aquellas que cumplan las restricciones dadas.

El mecanismo de clasificación de LOOM, que mantiene las jerarquías de conceptos y relaciones, facilita la definición de ciertos tipos de índices genéricos. Estos tipos de índices representan que en el cómputo de similitud se utilizarán únicamente las relaciones del dominio clasificadas bajo las relaciones de CBROnto especificadas en el conjunto de restricciones terminológicas. Un diseñador puede definir tipos de índices propios, más específicos, o utilizar alguno de los predefinidos, y asociarlos con cualquier concepto, incluyendo los conceptos del dominio, y los conceptos que representan los tipos de casos.

En base a la clasificación de tipos de relación descrita en el Apartado 4.1.1 en CBROnto existen los siguientes tipos de índice genéricos predefinidos:

- Índice *semántico* representado por el individuo `semantic-type` (instancia de `IndexType`). Tiene en cuenta todos los conceptos y relaciones del dominio que intervienen en la descripción del concepto, y es la que se utiliza por defecto:
  `index-restrictions '((binary-tuple Thing)))`

- Índice *estructural* representado por el individuo `composition-type` (instancia de `IndexType`). Tiene en cuenta únicamente las relaciones de composición (`composition`) como `part-of` y `has-part`: `(index-restrictions '((composition)))`
- Índice *espacial* representada por el individuo `spatial-type` (instancia de `IndexType`). Tiene en cuenta las relaciones espaciales (`spatial`) como `near_of`, `far` o `touching`: `(index-restrictions '((spatial)))`
- Índice *temporal* representado por el individuo `temporal-type` (instancia de `IndexType`). Utiliza las relaciones y conceptos clasificados como temporales (`temporal`) como `during`, `before` o `after`: `(index-restrictions '((temporal)))`
- Índice *causal* representado por el individuo `causal-type` (instancia de `IndexType`). Utiliza las relaciones causales `depends-on`, `cause` y `explain`: `(index-restrictions '((causing)))`
- Índice *funcional* representado por el individuo `function-type` (instancia de `IndexType`). Utiliza las relaciones `gets` y `has-precondition`, restringidas a los conceptos `Goal` y `Precondition`: `(index-restrictions '((gets Goal) (has-precondition Precondition)))`
- Índice *descriptivo* representado por el individuo `description-type` (instancia de `IndexType`). Utiliza las relaciones clasificadas como `description-property`: `(index-restrictions'((description-property)))`

En el ejemplo de la planificación del movimiento de bloques, si usamos un índice semántico recuperaremos el caso más similar teniendo en cuenta todos los aspectos que lo describen, tanto las características que describen a las piezas (incluyendo el tipo, posición y peso). Si utilizamos únicamente el tipo de índice estructural recuperaremos el caso más similar según las relaciones `has-part`, es decir, basándonos en la distribución final de las piezas en los contenedores, pero no según la similitud entre las características que describen a las piezas (`weigth` y `has-location`).

El hecho de que el diseñador asocie varios tipos de índice con un mismo concepto permite representar distintos modos de uso que serán ofrecidos al usuario final. En particular, entre los tipos de índices asociados al tipo de caso que representa la base de casos en la que vamos a recuperar. Si durante la fase de diseño no se definen tipos de índice asociados con los tipos de caso o de consulta, el sistema utilizará por defecto el tipo de índice semántico.

5.2.2 Estrategias de transformación de índices

Un índice usado para recuperar casos de la memoria puede fallar incluso si hay un caso relevante en la base de casos. Esto ocurrirá cuando el índice dado por la consulta actual no es el índice bajo el que está organizado el caso. Como la organización de la base de casos es estática, una posible solución sería modificar el índice obtenido a partir de la consulta para obtener un punto de vista diferente de la base de casos y permitir así la recuperación de casos que eran inaccesibles a partir del índice original. Se puede considerar la posibilidad de generar nuevos índices a partir de los existentes o de usar alguna técnica de transformación. Existen varias estrategias sencillas de transformación de índices [Sycara&Navichandra89] que permiten procesos de recuperación más flexibles que encuentren casos que no corresponden exactamente con la situación prevista a priori, es decir, que no están clasificados bajo los índices obtenidos a partir de la consulta. Por ejemplo, la *elaboración* consiste en añadir detalles a un índice existente. La *abstracción* trata de encontrar un índice más general al que pertenece el índice actual. La *mutación* se basa en sustituir o modificar componentes de la consulta y, por

último, la *generación* incrementa el alcance del caso objetivo usando técnicas deductivas para inferir conocimiento adicional sobre los casos.

CBROnto incluye un tipo de generación que se basa en la compleción de instancias de LOOM, que permite ayudar a construir consultas asociando condiciones suficientes a la definición de los conceptos que representan los tipos de consulta. Además de utilizarlo con los tipos de consulta, este mecanismo también se puede aprovechar en el resto de los conceptos del dominio: si una consulta se reconoce como una instancia de un concepto, el mecanismo de compleción de instancias de LOOM permite inferir nuevas características. Este mecanismo es el utilizado para incluir las reglas de dependencia extraídas por AFC (Apartado 5.3.3.2). Además, en CBROnto se ofrece una estrategia de abstracción que utiliza las jerarquías de términos que representan conocimiento del mundo para generalizar las propiedades proporcionadas por la consulta, es decir, generalizar los conceptos bajo los que se ha clasificado la consulta, o generalizar las relaciones que describen a la consulta.

Las opciones de indexación que hemos descrito en este apartado se basan principalmente en la intervención del diseñador que realizará una indexación manual de las relaciones o definirá conceptos índices para organizar la base de casos que reflejen los objetivos de la aplicación diseñada. El siguiente apartado describe una alternativa para organizar la base de casos basada en extraer información de modo inductivo a partir de los casos.

5.3 El Análisis Formal de Conceptos

Como hemos descrito en el Apartado 5.1, la definición de conceptos para organizar los casos enriquece el conocimiento del dominio y facilita el acceso y la recuperación de casos. Como alternativa a la definición manual de estos conceptos se puede optar por el uso de técnicas inductivas –como el Análisis Formal de Conceptos- para construir estos conceptos de forma automática a partir de los casos.

Desde un punto de vista general, la teoría del AFC permite la clasificación y la estructuración automática de la información mediante los *conceptos formales* de un dominio o contexto. El AFC tiene especial interés cuando es necesario trabajar con un gran número de entidades y objetos que pueden describirse mediante un conjunto de propiedades o atributos. Esto ocurre los sistemas de CBR donde los casos de la biblioteca se representan mediante un conjunto de propiedades o atributos.

En CBROnto utilizaremos el AFC para estructurar y clasificar los casos en una jerarquía de conceptos formales que representan las regularidades o asociaciones entre ellos según sus atributos. Como veremos en el Capítulo 5, esta jerarquía de conceptos de indexación permitirá la recuperación posterior de forma muy *eficiente* y *directa* de los casos más relevantes para la consulta del usuario.

5.3.1 Conceptos y contextos formales

El Análisis Formal de Conceptos [Davey&Priestley90] [Wille92] [Ganter&Wille97] es una teoría de formación de conceptos derivada de la teoría de retículos y conjuntos ordenados que proporciona un modelo matemático para la construcción y análisis de jerarquías conceptuales. Desde el punto de vista informático, el AFC es una técnica automática para la estructuración y clasificación de la información que puede utilizarse para encontrar patrones, regularidades, excepciones, etc. haciendo visible y accesible la estructura conceptual de la información.

La idea básica en la que se fundamenta el AFC es la noción de *concepto formal*. Un concepto formal está definido por dos partes: su *extensión* y su *intensión*. La extensión de un concepto consiste en el conjunto de objetos o instancias de ese concepto (por ejemplo, para el concepto persona cada una de las personas concretas), y la intensión es la colección de todos los atributos (o propiedades) que tienen en común los objetos que son instancias del concepto (por ejemplo, todas las personas comparten como característica común que pueden respirar). Como ocurre siempre que se crea un modelo informático de un dominio real, se trabaja dentro de un contexto específico en el que el conjunto de objetos y atributos está previamente definido y limitado. El modelo matemático que representa los objetos, los atributos y la relación entre ellos se denomina *contexto formal* y se define como una terna (G,M,I) formada por dos conjuntos G y M[3] de objetos y atributos, respectivamente, y una relación binaria $I \subseteq G \times M$ entre ellos. Los elementos de G ($g \in G$) representan los objetos o entidades del contexto, mientras que los elementos de M ($m \in M$) representan los atributos o características que los objetos pueden tener asociados. La relación gIm (o $(g,m) \in I$) significa que el objeto g tiene el atributo m (o que m se aplica a g).

Para $A \subseteq G$ y $B \subseteq M$ definimos los conjuntos A' y B' como:

$\mathbf{A'} = \{m \in M \mid (\forall g \in A)\ gIm\}$

$\mathbf{B'} = \{g \in G \mid (\forall m \in B)\ gIm\}$

A' es el conjunto de todos los atributos de M que se aplican a todos y cada uno de los objetos de A, y B' es el conjunto de objetos de G que tienen todos los atributos de B.

Def. Un *concepto formal* del contexto (G,M,I) se define como un par extensión-intensión (A,B) donde $A \subseteq G$ y $B \subseteq M$, que cumple que A'=B y B'=A. Además también se cumplen las igualdades A''=A y B''=B, es decir:

- B está formado (es decir contiene todos y ninguno más) de los atributos pertenecientes a M que se aplican sobre todos los objetos de A.
- A está formado por los objetos de G que tienen todos los atributos de B.

5.3.2 Orden conceptual y retículos de conceptos

Sobre el conjunto de todos los conceptos formales de un contexto (G,M,I), que se denota por $\mathcal{B}(G,M,I)$, se define la siguiente *relación de orden* ($\leq$):

Def. Dados dos conceptos formales c1 = (A1, B1) y c2 = (A2, B2) pertenecientes a $\mathcal{B}(G,M,I)$, se dice que $c1 \leq c2$, c1 es un subconcepto de c2 (o que c2 es un superconcepto de c1) si $A1 \subseteq A2$ (lo que es equivalente a que $B2 \subseteq B1$)

Lo que intuitivamente quiere decir que un concepto siempre tiene una extensión más "pequeña" y una intensión más "grande" que cualquiera de sus superconceptos.

La relación ($\leq$) cumple las propiedades *reflexiva*, *transitiva* y *antisimétrica*, por lo tanto es una relación de orden, y hace que $\mathcal{B}(G,M,I)$ sea un conjunto ordenado. Además, como enuncia el teorema fundamental del AFC, $\langle \mathcal{B}(G,M,I) ; \leq \rangle$ es un *retículo completo*, y se conoce como *retículo de conceptos del contexto (G,M,I)*. Antes de enunciar el teorema fundamental del AFC, repasamos dos definiciones básicas:

Def. Se dice que un conjunto ordenado (P, $\leq$) es un **retículo** cuando para cualquier par de elementos pertenecientes a P existen el supremo y el ínfimo.

[3] Del alemán Gegenstände y Merkmale

Def. Se dice que un conjunto ordenado (P, ≤) es un **retículo completo** si para cualquier subconjunto S de P (S⊆P), existen el ínfimo (Inf S) y el supremo (Sup S).

5.3.2.1 Teorema Fundamental del AFC

En [Wille 82] se enuncia el teorema fundamental del Análisis Formal de Conceptos:

Sea (G,M,I) un contexto. Entonces el conjunto $\mathcal{B}(G,M,I)$ de todos los conceptos formales pertenecientes al contexto es un *retículo completo* en el que se pueden formalizar los ínfimos (∧) y los supremos (∨) como:

$$\text{Inf } \mathcal{B}(G,M,I) \equiv \bigwedge_{\alpha}(A_\alpha, B_\alpha) = \left[\bigcap_{\alpha} A_\alpha, \left(\bigcup_{\alpha} B_\alpha\right)''\right]$$

$$\text{Sup } \mathcal{B}(G,M,I) \equiv \bigvee_{\alpha}(A_\alpha, B_\alpha) = \left[\left(\bigcup_{\alpha} A_\alpha\right)'', \bigcap_{\alpha} B_\alpha\right]$$

El ínfimo (*meet*) de un conjunto de conceptos es un subconcepto que se construye como una especialización de todos ellos. Se corresponde con el mayor concepto que es subconcepto de todos los conceptos de partida. La extensión de este subconcepto está formada por los objetos comunes a las extensiones de los conceptos de los que se parte, mientras que su intensión está formada por los atributos comunes al conjunto de objetos.

El supremo (join) de un conjunto de conceptos es una generalización de dichos conceptos que se corresponde con el concepto más específico que es superconcepto de todos los conceptos de partida. El conjunto de atributos de este superconcepto se encuentra formado por los atributos comunes a las intensiones de los conceptos de partida y su extensión está formada por los objetos que tengan dichos atributos.

Además, para cualquier conjunto ordenado ⟨S, ≤⟩ se puede definir una *relación de cobertura* (≺) entre sus elementos:

Def. Sean $x_1, x_2 \in S$. Decimos que x_2 cubre a x_1 y escribimos $x_1 \prec x_2$, si y sólo si se cumple: [Davey&Priestley90]

(i) $x_1 \leq x_2$ y $x_1 \neq x_2$

(ii) $\forall\, x_3 \in S$, si $x_1 \leq x_3 \leq x_2$ entonces $x_1 = x_3$ o $x_3 = x_2$

Obsérvese que, si S es finito, $x \leq y$ si y sólo si existe una secuencia finita de relaciones ≺ de la forma $x = x_0 \prec x_1 \prec .... \prec x_n = y$. Así, en el caso finito, la relación de orden determina, y es determinada por, la relación de cobertura.

Podemos representar el conjunto ordenado $\mathcal{B}(G,M,I)$ mediante un *diagrama de Hasse* en el que los nodos representan a los elementos del conjunto (en este caso los conceptos formales) y las líneas entre los nodos representan la relación de cobertura. Si un elemento c2 cubre a otro elemento c1 (c1≺c2), el elemento c2 se representa por encima de c1 y quedan unidos por una línea que, además, no pasa por ningún otro elemento del conjunto.

Podemos encontrar algunos puntos en común entre el AFC y las DLs como mecanismo de representación de conocimiento, lo que ha facilitado la integración de las dos técnicas en nuestro trabajo. Por ejemplo, el lenguaje de definición de conceptos de las DLs permite,

como característica básica, la definición de clases de individuos (objetos) que comparten una serie de propiedades (atributos). Además, ambas comparten la idea de estructurar los conceptos en una jerarquía de especializaciones, o *taxonomía*, con el concepto genérico en la raíz y los conceptos más específicos en las hojas. De esta forma, el *join* o supremo de un conjunto de conceptos se corresponde directamente con el LCS (mínimo límite superior) en la terminología de las DLs y, de manera análoga, el *meet* o ínfimo se corresponde con el MLI (mínimo límite inferior).

5.3.3 Usos del AFC para el CBR

Nuestra propuesta es la aplicación del AFC para extraer el conocimiento de una base de casos, en particular las dependencias entre los atributos que describen los casos. Este conocimiento se utilizará para crear un sistema de índices que completará el conocimiento terminológico del dominio adquirido mediante otras técnicas. La aplicación del AFC a una base de casos permite obtener una estructura de organización de casos, ya que extrae los conceptos formales y las relaciones jerárquicas entre ellas, donde los casos se agrupan según las propiedades que comparten. Además el conjunto de reglas de dependencia extraído de los casos se utiliza para guiar el proceso de formulación de consultas para la recuperación de casos sobre el retículo [Díaz&González00b] [Díaz&González01a] [Díaz&González01c].

5.3.3.1 Construcción del retículo de conceptos formales

El primer paso para poder aplicar AFC a una base de casos, es interpretar la misma como un contexto formal. Normalmente una base de casos se interpreta como un contexto formal multivaluado, es decir, definido como una estructura (G, M, W, I) donde G es un conjunto de objetos, M es un conjunto de atributos, W es un conjunto de valores de atributos e I es una relación ternaria entre G, M y W ($I \subseteq G \times M \times W$); $(g,m,v) \in I$ significa que el objeto g tiene el valor v en el atributo m, y si se cumple $(g,m,v) \in I$ y $(g,m,w) \in I$ entonces siempre ocurre que $v=w$. Esta propiedad inhibe la aplicación del AFC en atributos con más de un relleno.

Para facilitar la comprensión de la técnica del AFC, utilizaremos un primer ejemplo sencillo sobre una base de casos en el dominio de los viajes, pero simplificada a una representación de casos plana utilizando vectores de pares atributo-valor. El apartado siguiente explica como extender la aplicación del AFC a casos con estructuras complejas. La Tabla 4-1 representa una base con siete casos que puede interpretarse como un contexto multivaluado en el que G es el conjunto de casos (filas), M es el conjunto de atributos (columnas) y W es el conjunto que contiene todos los valores de los atributos que aparecen en las celdas.

	DESTINO	TIPO	TRANSPORTE	ESTACIÓN
Caso 1	España	Educación	Coche	Verano
Caso 2	Nueva York	Esquí	Avión	Invierno
Caso 3	España	Educación	Tren	Verano
Caso 4	Inglaterra	Activas	Coche	Otoño
Caso 5	España	Esquí	Avión	Invierno
Caso 6	Inglaterra	Educación	Tren	Verano
Caso 7	Nueva York	Activas	Avión	Primavera

Tabla 4-1. *Representación de casos simplificada*

Un contexto multivaluado puede transformarse a un contexto simple (univaluado) aplicando una escala transformacional (*transformational scaling)* [Wille92][Ganter& Wille97] [Prediger&Stumme99]. En nuestro ejemplo aplicamos la más simple de las transformaciones (*plain scaling*) en la que cada atributo del contexto multivaluado inicial se sustituye por un conjunto de columnas que representan cada uno de los valores del atributo. De esta forma obtenemos quince columnas que sustituyen las cuatro columnas de la tabla anterior. Las cruces en la tabla indican que un objeto tiene un atributo, o lo que es lo mismo, que una cierta característica se usa como un descriptor de un caso. En la Tabla 4-2 los nombres de los atributos usando únicamente las letras iniciales.

	D::España	D::Egipto	D::Inglaterra	D::Nueva York	HT::Activas	HT::Educación	HT::Language	HT::Esquí	T::Coche	T::Avión	T::Tren	S::Verano	S::Invierno	S::Otoño	S::Primavera
Caso 1	☑					☑			☑			☑			
Caso 2				☑				☑		☑			☑		
Caso 3	☑					☑					☑	☑			
Caso 4			☑		☑				☑					☑	
Caso 5	☑							☑		☑			☑		
Caso 6			☑			☑					☑	☑			
Caso 7				☑	☑					☑					☑

Tabla 4-2. *Relación de incidencia*

Obviamente, en el proceso de transformación se pierde cierto conocimiento ya que los atributos escalados son independientes entre sí, es decir, existe la misma relación entre `Destino::España` y `Destino::Egipto` que entre `Destino::España` y `Transporte::Coche`. Aunque esto no supone un problema importante en nuestra aproximación, ya que el AFC es una técnica complementaria y no la técnica básica de representación de conocimiento, se puede extender esta aproximación y utilizar escalas conceptuales más complejas. En concreto, es útil que la selección de los atributos de la escala refleje el conocimiento de un experto del dominio. Esta aproximación es utilizada en nuestro esquema para los atributos numéricos, en los que los valores resultan demasiado específicos para realizar la escala y son sustituidos por atributos más generales, por ejemplo por la propiedad binaria de pertenencia a ciertos intervalos que reflejen el punto de vista de un experto del dominio.

La Tabla 4-2 representa un contexto formal (G,M,I) definido por los conjuntos G (de objetos) y M (de atributos) y la relación binaria $I \subseteq G \times M$ entre ellos. A partir del contexto formal (G,M,I) se aplica el teorema fundamental del AFC para determinar los elementos del conjunto $\mathcal{B}$ (G,M,I), es decir, los conceptos formales del contexto. En la Tabla se muestran los conceptos obtenidos, descritos mediante su *extensión* y su *intensión*.

Además de la representación tabular, existe una representación gráfica de los contextos formales. En la Figura 4-19 se muestra el diagrama de *Hasse* correspondiente al conjunto de conceptos formales de la Tabla 4-2. Cada nodo del diagrama representa un concepto formal del contexto, y los caminos ascendentes de aristas representan la relación subconcepto-superconcepto. Los conceptos formales dentro del diagrama de *Hasse* cumplen que para cada dos nodos cualesquiera c1=(A1,B1), c2=(A2,B2), si $c1 \leq c2$, es decir, existe un camino ascen-

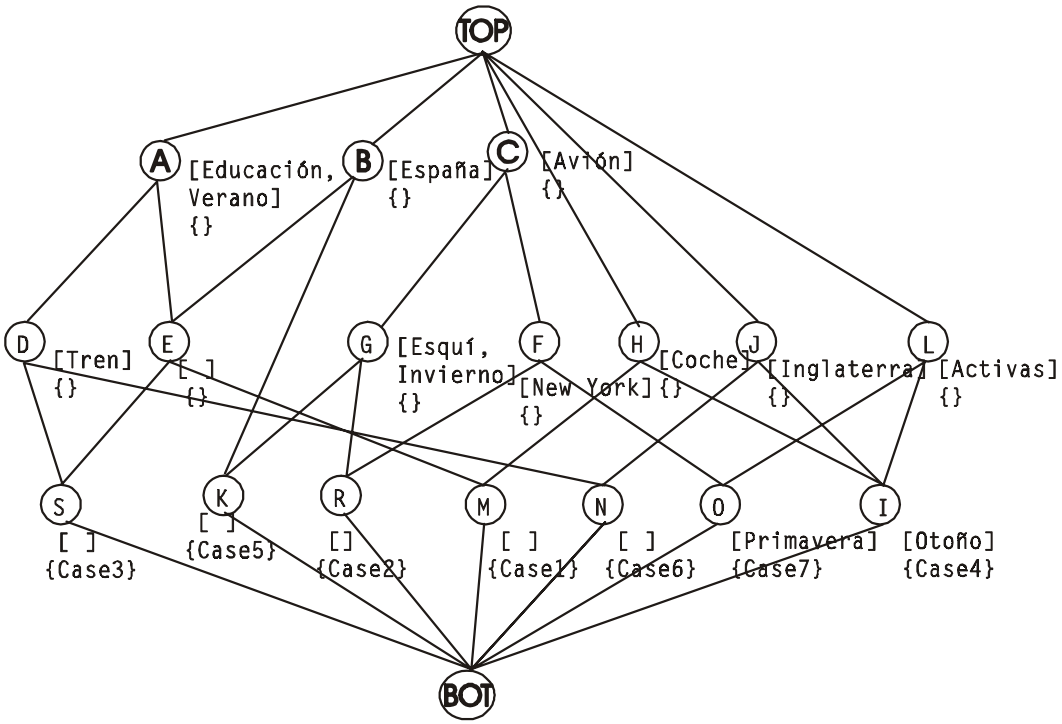


Figura 4-19. *Retículo de conceptos formales*

dente desde c1 hasta c2, entonces A1⊆A2 y B2⊆B1, la extensión de c1 está contenida en la de c2 y la intensión de c2 está contenida en la de c1.

Cada nodo del diagrama de Hasse se etiqueta para indicar los objetos y los atributos que lo definen. Los nombres de los objetos se escriben entre {} y los atributos entre [ ] incluyéndolos sólo en los nodos generados a partir de ellos. Un nodo se etiqueta con un atributo $m \in M$ si es el nodo más general que contiene a m en su intensión, y un nodo se etiqueta con el objeto $g \in G$ si es el nodo más específico que contiene a g en su extensión. Con esta forma de etiquetado (que se usa por convenio) cada etiqueta, tanto de atributo como de objeto, aparece exactamente una vez en el diagrama. Si un nodo C está etiquetado con el atributo [m] y el objeto {g} entonces todos los conceptos mayores que C (es decir, que están encima de C en el diagrama) tienen el objeto g en sus extensiones, y todos los conceptos menores que C (debajo de C en el diagrama) tienen el atributo m.

Para cada concepto del retículo podemos leer su extensión como la unión de los objetos que aparecen en su etiqueta {} y los que aparecen en las etiquetas {} de todos sus subconceptos. De forma inversa para leer su intensión tomamos la unión de los atributos del concepto (etiqueta [ ]) y los de sus superconceptos (caminos ascendentes en el retículo). En la Tabla 4-3 se muestra la extensión y la intensión completas para todos los conceptos formales del diagrama de la Figura 4-19

Es importante observar que el retículo contiene exactamente la misma información que la tabla que representa la relación de incidencia ya que ésta siempre se puede reconstruir a partir del retículo. Para reconstruir una fila de la relación de incidencia original de la tabla, se busca el único concepto C cuya etiqueta {} contiene el nombre del objeto correspondiente a la fila que queremos reconstruir y marcamos las columnas de los atributos que forman la intensión

B (G,M,I) : Conjunto de conceptos formales		
Concepto Formal	**Extensión (número de los casos)**	**Intensión**
TOP	1,2,3,4,5,6,7	∅
A	1,3,6	Educación, Verano
B	1,3,5	España
C	2,5,7	Avión
D	3,6	Educación, Tren, Verano
E	1,3	España, Educación, Verano
F	2,7	NewYork, Avión
G	2,5	Esquí, Avión, Invierno
H	1,4	Coche
J	6,4	Inglaterra
L	7,4	Activas
M	1	España,Educación,Coche,Verano
R	2	NewYork, Esquí, Avión, Invierno
S	3	España, Educación, Tren, Verano
I	4	Inglaterra,Activas,Coche,Otoño
K	5	España, Esquí, Avión, Invierno
N	6	Inglaterra, Educación, Tren, Verano
O	7	NewYork, Activas, Avión, Primavera
BOT	∅	España, Inglaterra, NewYork, Esquí, Educación, Activas, Avión, Coche, Tren, Verano, Invierno, Otoño, Primavera

***Tabla 4-3**. Extensión e intensión de los conceptos formales*

de C. Por ejemplo, para reconstruir la fila del Caso 6 marcaremos las columnas correspondientes a la intensión del concepto N: `[Tren, Educación, Verano, Inglaterra]`. De forma dual, para reconstruir una columna de la tabla, se busca el concepto C cuya etiqueta [ ] contiene el nombre de atributo que corresponde a la columna y marcaremos aquellas filas correspondientes a cada uno de los objetos de su extensión. Por ejemplo, para reconstruir la columna `Verano` marcaremos las filas correspondientes a la extensión del concepto A: `{Caso3, Caso1, Caso6}`.

5.3.3.2 Extracción de las reglas de dependencia

Además de la clasificación conceptual de los casos, el retículo de conceptos formales proporciona un conjunto de implicaciones entre los atributos, a las que llamaremos reglas de dependencia. Una regla de dependencia entre dos conjuntos de atributos se escribe $M1 \rightarrow M2$, siendo $M1,M2 \subseteq M$, y significa que cualquier objeto que tenga todos los atributos de $M1$ tiene también todos los atributos de $M2$.

En un retículo de conceptos formales etiquetado de la forma descrita se pueden leer las reglas de dependencia de la siguiente forma:

- Cada línea entre dos nodos que estén etiquetados con conjuntos de atributos M1 y M2, significa una regla de dependencia entre los dos conjuntos de atributos del nodo más específico al más general M1 → M2.
- Si hay varios atributos en la misma etiqueta significa que dichos atributos coaparecen en todos los casos. La idea subyacente es que si existen dos propiedades P y Q que

coaparecen en todos los casos de la base: P(c1) ∧ Q(c1); P(c2) ∧ Q(c2) se lleva a cabo un proceso de generalización que da lugar a las reglas ∀x. P(x) → Q(x) y ∀x. Q(x) → P(x). Así, durante el proceso de formulación de consultas se sabe P(cq) se infiere Q(cq) por instanciación de esta generalización [Davies&Russell87].

Por ejemplo, la etiqueta de dos atributos `[Esquí, Invierno]` significa que en esta base de casos todos los viajes a esquiar se ofrecen en invierno y viceversa, que todos los viajes en invierno son a esquiar. Es decir, se induce una regla de dependencia bidireccional de la forma `Esquí ↔ Invierno`, que se divide en las dos reglas de dependencia: `Esquí → Invierno y Invierno → Esquí`.

Aplicando lo anterior se puede leer del diagrama el siguiente conjunto de dependencias:

`{ Tren → Educación, Verano;Nueva York → Avión; Esquí, Invierno → Avión ; Primavera → Nueva York; Otoño → Coche; Otoño → Inglaterra; Otoño → Activas; Esquí → Invierno; Invierno → Esquí ; Educación → Verano; Verano → Educación }`

Este conjunto de implicaciones entre los atributos es conocimiento de esta base de casos en concreto y se utiliza para completar el modelo del dominio y para guiar el proceso de formulación de consultas sobre esta base de casos.

5.3.3.3 Aplicación del AFC en casos con objetivos y precondiciones

En el apartado anterior hemos visto que la aplicación del AFC facilita que una vez identificados los índices de los casos, es decir, las características relevantes, éstos puedan organizarse de manera que el proceso de localización de casos similares se optimice. El primer paso para poder aplicar AFC a una base de casos es interpretar la misma como un contexto formal, y por tanto, determinar qué atributos (o relaciones) de los casos se utilizan como índices, y forman el conjunto de atributos del contexto formal (conjunto W).

En este apartado describimos una aplicación particular del AFC cuando los casos incluyen una representación explícita de los objetivos que consiguen y/o de las precondiciones que se deben satisfacer para poder aplicar el caso. Supongamos que en el dominio de aplicación se han identificado ciertos objetivos básicos A,B,C,D,E (conceptos clasificados como GOAL) y ciertas propiedades observables X, Y, Z, T (conceptos clasificados como PROPERTY). Además existirán acciones que se llevan a cabo sobre entidades del dominio. Para simplificar la exposición, suponemos también que todos los objetivos y propiedades son primitivos (y no definidos) y que para cada caso concreto se asertan (y no se infieren) explícitamente los objetivos que satisface la solución. Esto no será así en una situación general, en la que se puede inferir los objetivos (conceptos) que satisface un caso en función de las acciones de su solución como en el ejemplo del movimiento de bloques. Suponemos casos de tipo funcionalidad para los que su solución se relaciona (a través de sus entidades) con las precondiciones necesarias para su aplicación y con los objetivos que satisface.

Si suponemos que la base de casos para el ejemplo está formada por los seis casos siguientes, y que se plantea una consulta en la que se quieren obtener los objetivos A,B,C y la situación actual se describe por las propiedades X,Y,Z.

	Precondiciones	Objetivos
Caso1	X,Y	A,B,C
Caso2	X,Y,Z	A,C
Caso3	Z	A,B,D,E
Caso4	Y,Z	A,B
Caso5	X,Y,Z,T	D,E
Caso6	Y,T	A,C

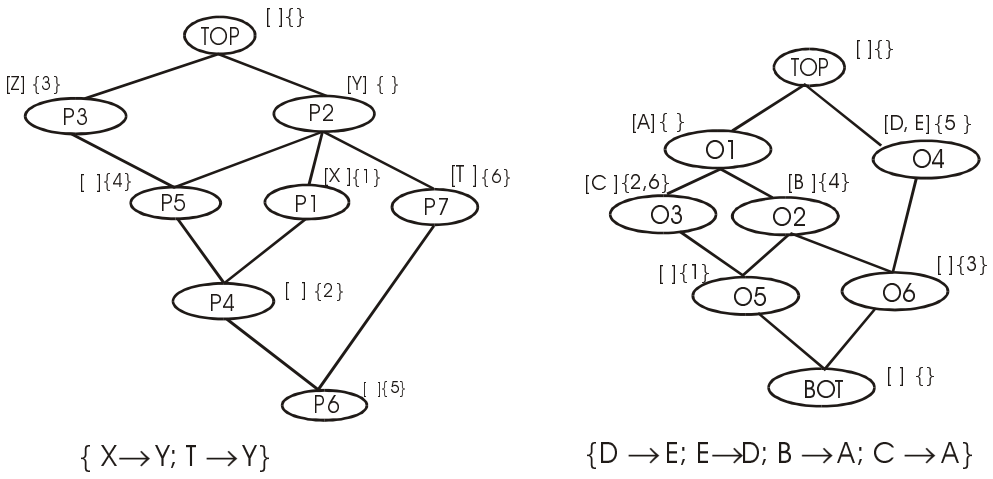


Figura 4-20. *Retículos de conceptos formales de precondiciones (izquierda) y objetivos (derecha)*

Los casos aplicables, es decir, aquellos para los que se satisface su precondición y por tanto podríamos aplicar su solución, son Caso1, Caso2, Caso3 y Caso4. Además, Caso1 es el único que satisface todos los objetivos pedidos, por lo que será recuperado como el mejor caso y se aplicará su solución.

Aplicaremos la técnica del AFC para obtener dos retículos de conceptos formales, uno para las propiedades y otro para los objetivos, que serán recorridos de forma independiente durante la recuperación (este método se describe en el Capítulo 5-Apartado 3.2.4.3). El retículo de objetivos captura conceptos formales que representan la coaparición de objetivos resueltos por los casos, y el retículo de precondiciones captura la coaparición de propiedades en las precondiciones.

Para la aplicación del AFC, interpretamos la aparición de objetivos y precondiciones de los casos como contextos formales que se representan mediante las relaciones de incidencia representadas en la Tabla 4-4, donde Prei es un individuo que representa todas las precondiciones que requiere el Casoi para ser aplicado, y Goali es un individuo que representa la conjunción de objetivos que se satisfacen después de ejecutar la solución del Casoi.

	X	Y	Z	T
Pre1	☑	☑		
Pre2	☑	☑	☑	
Pre3			☑	
Pre4		☑	☑	
Pre5	☑	☑	☑	☑
Pre6		☑		☑

	A	B	C	D	E
Goal1	☑	☑	☑		
Goal2	☑		☑		
Goal3	☑	☑		☑	☑
Goal4	☑	☑			
Goal5				☑	☑
Goal6	☑		☑		

Tabla 4-4. *Relaciones de incidencia de precondiciones y objetivos*

La Figura 4-20 muestra los diagramas de Hasse para los retículos de conceptos formales asociados con los contextos de precondiciones y objetivos, y los conjuntos de reglas de de-

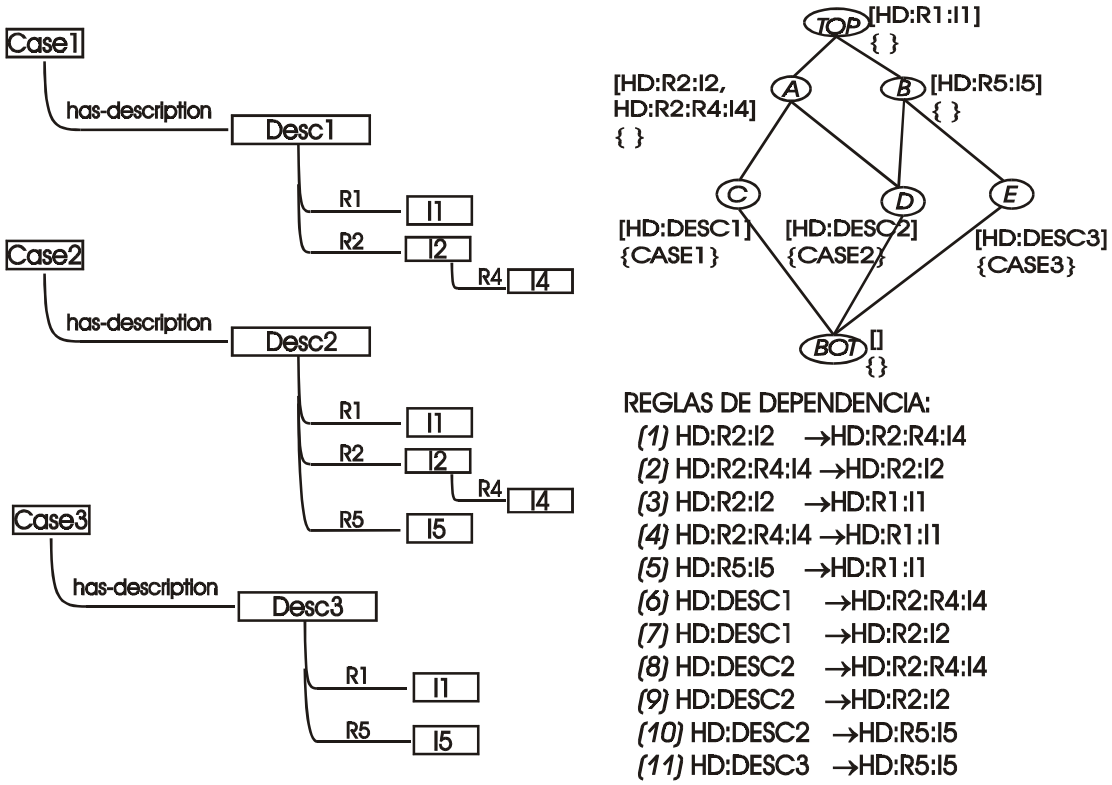


***Figura 4-21**. Aplicación del AFC en casos estructurados*

	HD:**DESC1**	HD:**DESC2**	HD:**DESC3**	HD:R1:**I1**	HD:R2:**I2**	HD:R5:**I5**	HD:R2:R4:**I4**
CASE1	☑			☑	☑		☑
CASE2		☑		☑	☑	☑	☑
CASE3			☑	☑		☑	

***Tabla 4-5.** Relación de incidencia en casos estructurados*

pendencias que se pueden extraer de cada uno de ellos, que son específicas de esta base de casos (y no reglas generales del dominio).

5.3.3.4 Aplicación de AFC en casos estructurados

En los dos ejemplos anteriores los casos son descritos mediante una estructura plana, en la que todos ellos incluyen el mismo conjunto de características. En este apartado describimos cómo aplicar el AFC en bases de casos estructurados o parcialmente definidos.

Cuando se utilizan atributos relacionales para describir casos con estructuras complejas construimos los conjuntos *G* y *M* como sigue: *G* es el conjunto de casos y *M* es el conjunto de descriptores de los casos, y se construye sobre el grafo de representación de cada caso, nivel por nivel, obteniendo los caminos parciales de relaciones que llevan desde el individuo

***B* (G,M,I) : conjunto de conceptos formales**		
Concepto Formal	**Extensión**	**Intensión**
TOP	Case1, Case2, Case3	[hd:R1:**I1]**
A	Case1, Case2	[hd:R1:**I1]** [hd:R2:**I2]** **[**hd:R2:R4:**I4]**
B	Case2, Case3	[hd:R1:**I1]** [hd:R5:**I5]**
C	Case1	[hd:**desc1**][hd:R1:**I1]**[hd:R2:**I2]** **[**hd:R2:R4:**I4]**
D	Case2	[hd:**desc2]** [hd:R1:**I1]** [hd:R5:**I5]** [hd:R2:**I2]** **[**hd:R2:R4:**I4]**
E	Case3	[hd:**desc3]** [hd:R1:**I1]** [hd:R5:**I5]**
BOT	∅	[hd:**desc1**] [hd:**desc2]** [hd:**desc3**][hd:R1:**I1]** [hd:R5:**I5]** [hd:R2:**I2]** **[**hd:R2:R4:**I4]**

***Tabla 4-6**. Extensión e intensión de los conceptos formales*

caso hasta individuos (que pueden ser terminales o no). Cada descriptor en *M* incluye un camino (posiblemente incompleto) de relaciones y el individuo (interno o terminal) que se alcanza desde el individuo caso al seguir el camino de relaciones.

La Tabla 4-5 muestra los conjuntos *G* y *M* (los nombres de las filas y las columnas respectivamente) para el conjunto de 3 casos cuya estructura de representación se muestra en la Figura 4-21 (izquierda). HD abrevia la relación `has-description`. La Figura 4-21 muestra el retículo de conceptos formales y el conjunto de reglas de dependencia que resultan de la aplicación del AFC. La Tabla 4-6 muestra la intensión y la extensión de cada concepto formal que se puede leer sobre este retículo.

En este apartado hemos descrito la técnica del AFC y cómo aplicarla a una base de casos para extraer una estructura conceptual de organización de estos casos y un conjunto de reglas de dependencia entre los descriptores de los casos. En el capítulo siguiente describiremos los métodos que hacen uso de este conocimiento, en concreto, para guiar la formulación de consultas y para el proceso de recuperación de casos.

6. Resumen y conclusiones del capítulo

En este capítulo hemos abordado nuestra ontología de CBR, CBROnto, desde distintas perspectivas. CBROnto incluye terminología CBR en general y conocimiento sobre la resolución de problemas mediante CBR. CBROnto está constituida por un conjunto de conceptos, relaciones e instancias implementados en LOOM. Para recapitular sobre qué tipo de conocimiento podemos encontrar en nuestra ontología diremos que:

- Incluye terminología CBR en general.
- Incluye conocimiento sobre CBR, por ejemplo tipos de CBR, tipos de casos y las características de cada uno.
- Incluye la terminología con la que se define un lenguaje de representación de casos.
- Incluye la terminología con la que se define un lenguaje de especificación de métodos PSMs.

- Incluye una ontología de tareas CBR independiente de un dominio de aplicación concreto.
- Incluye la formalización explícita de una biblioteca de métodos asociados y organizados en torno a las tareas CBR.
- Incluye los términos involucrados en los métodos de la biblioteca. Es decir, proporciona las definiciones de los conceptos y relaciones utilizados para especificar los métodos que resuelven las tareas CBR.

Obviamente, algunas de las facetas de CBROnto se solapan y no existe una división clara de qué términos pertenecen a cada una. El objetivo es que CBROnto sirva de marco general de ayuda para los diseñadores de aplicaciones CBR. Podemos considerar en cierto modo a CBROnto como una ontología de representación de conocimiento que, en vez de incluir primitivas de un lenguaje de representación de conocimiento general, incluye primitivas de un lenguaje para representar conocimiento sobre CBR, es decir, sobre sus casos, métricas de similitud, tipología, etc.

También se puede considerar que CBROnto es una ontología genérica ya que su conocimiento sobre CBR se puede reutilizar en varios dominios. En los sistemas de DLs parte de la semántica de un término la proporcionan los términos por encima de él en la jerarquía de subsunción, es decir, los términos de los que hereda información. En este sentido la clasificación de un término del dominio como especialización de un término de CBROnto proporciona la definición del término del dominio desde el punto de vista del CBR. En concreto se amplía la semántica del término para que pueda utilizarse en una aplicación CBR.

Además de describir las características principales de CBROnto, en este capítulo hemos introducido el Análisis Formal de Conceptos como una técnica para clasificar y estructurar la información de un dominio y que proporciona un modelo matemático para la construcción y el análisis de jerarquías conceptuales. La potencia de esta técnica viene dada por su capacidad para extraer de forma automática los conceptos relacionados con un contexto formal determinado. Los conceptos formales representan patrones y regularidades sobre el dominio, haciendo visible y accesible la estructura conceptual de la información. El AFC es adecuado cuando se va a trabajar con un gran número de entidades descritas mediante un conjunto de propiedades o atributos. Este modo de representar entidades también es adecuado a los sistemas de representación del conocimiento basados en DLs. Uno de los objetivos de este capítulo ha sido introducir los elementos básicos del AFC para extraer conocimiento de una base de casos que se utilizará posteriormente en los procesos de razonamiento de los sistemas CBR, en concreto, para la formulación de consultas y la recuperación de casos.

Capítulo 5
LOS MÉTODOS DE CBRONTO

1. Introducción

En el capítulo anterior hemos introducido CBROnto, una ontología que incluye terminología y conocimiento sobre la resolución de problemas mediante CBR. En particular, CBROnto incluye una biblioteca de métodos genéricos que resuelven las tareas involucradas en los sistemas CBR. En este capítulo detallamos cada uno de estos métodos.

Partiendo de la base de que nuestra biblioteca de métodos es limitada y, por supuesto ampliable en el futuro, hemos incluido en ella algunos métodos que son representativos de los sistemas KI-CBR y que sacan partido del conocimiento general sobre el dominio del que disponen los sistemas diseñados. Además, los métodos aprovechan los mecanismos de razonamiento de LOOM, el sistema de DLs en el que se basa nuestra propuesta.

En primer lugar describimos el proceso de resolución de tareas de COLIBRI con el objetivo de enmarcar el uso de los métodos en el proceso de diseño de una aplicación CBR. El proceso de diseño de aplicaciones CBR usando COLIBRI se detallará en el Capítulo 6. El resto del capítulo está organizado en torno a las tareas resultantes de aplicar el método CBR: recuperación, adaptación, revisión y aprendizaje.

Para cada método de la biblioteca asociado con alguna de estas tareas describimos su comportamiento general, las subtareas que genera y distintas opciones de métodos que resuelven estas subtareas. Además, para cada método hacemos hincapié en los requisitos que determinan su aplicabilidad. La formalización de los métodos como individuos representados en LOOM se incluye como parte del Apéndice B. Además, el Apéndice C incluye un listado textual de todos los métodos de la biblioteca. El Capítulo 6 incluye ejemplos de uso de los métodos de la biblioteca.

2. Resolución de tareas en COLIBRI

COLIBRI define un proceso de resolución de tareas general y recursivo que parte de una tarea y accede a los métodos elegidos y configurados por el diseñador durante la fase de diseño de la aplicación (que se describe en el Capítulo 6). Si el método es de resolución se usa

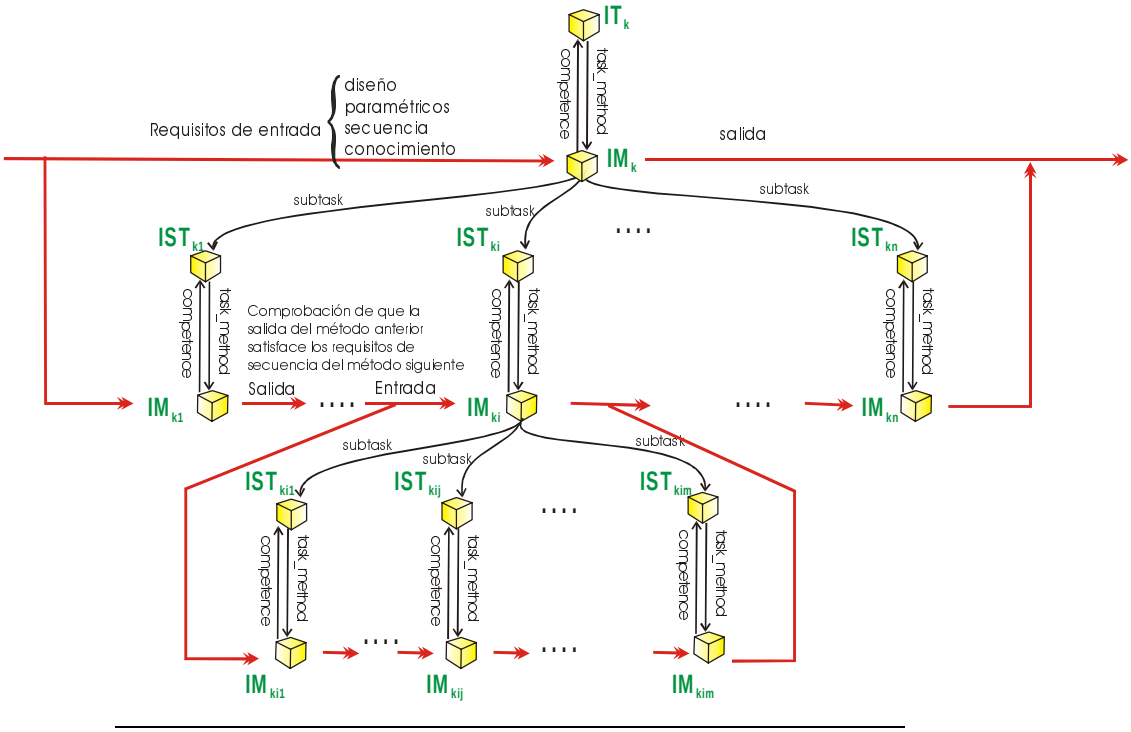


Figura 5-1. *El proceso de resolución de tareas de COLIBRI*

directamente para resolver la tarea y si es un método de descomposición divide la tarea en subtareas. Para cada una de las subtareas obtenidas aplicamos el proceso de resolución de forma recursiva.

Como vimos en el Capítulo 4, para cada método se especifican, entre otras cosas, la salida que genera, sus requisitos de entrada –que se descomponen en: requisitos de diseño, de secuencia, paramétricos y de conocimiento– y sus requisitos de aplicabilidad.

En la Figura 5-1 se muestra cómo se propagan las entradas y salidas de los métodos durante el proceso de resolución de tareas, a través de los distintos niveles de descomposición de tareas. Los requisitos de secuencia –que forman parte de los requisitos de entrada– describen las entradas que cada método espera recibir del método que resuelve la tarea anterior en una secuencia de resolución de tareas.

El proceso de resolución accede al método elegido para resolver cada tarea de una secuencia de tareas IST_{k1} …IST_{kn}–en la fase de diseño de la aplicación se establece la relación `task_method` entre la tarea y uno de los métodos que tienen una competencia (relación `competence`) adecuada. Dicha secuencia de tareas se genera al aplicar un cierto método de descomposición IM_k. La entrada del método IM_k se propaga al método que resuelve la primera subtarea de la secuencia –IST_{k1}– que en la Figura 5-1 es el método IM_{kn}. La salida del método IM_k se corresponde con la salida del método que resuelve la última subtarea de la secuencia –IST_{kn}– que en la figura es el método IM_{k1}. En una secuencia de métodos –IM_{k1} … IM_{kn}– que resuelve una secuencia de tareas – IST_{k1} …IST_{kn}– la salida del método IMki debe satisfacer los requisitos de secuencia del método IM_{ki+1}. Por su parte el método puede tener otras entradas externas (requisitos de diseño, parámetros o conocimiento del dominio).

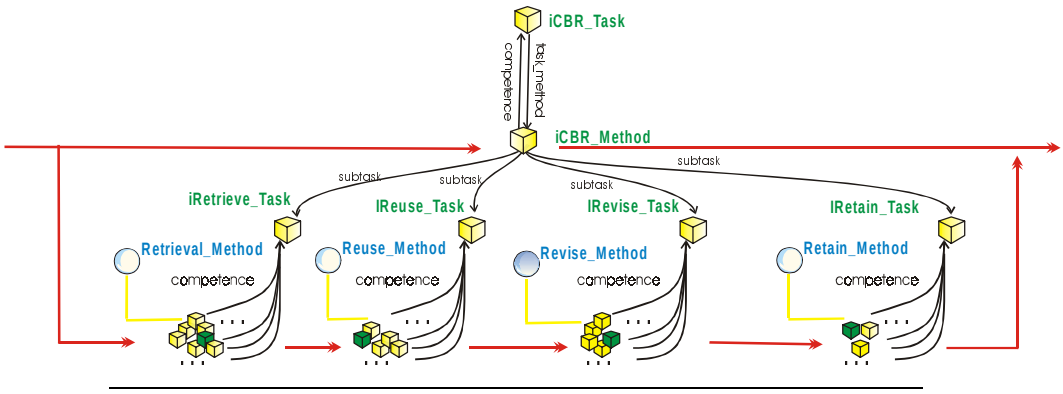


Figura 5-2. *Resolución de la tarea principal de un ciclo CBR*

Como veremos en el Capítulo 6, cada vez que el usuario final inicia un ciclo de resolución de una consulta, el sistema hará una llamada al resolutor de tareas con la tarea principal de resolución de problemas –copia del individuo canónico `iCBR_Task`. En nuestra biblioteca existe un único método que resuelve esta tarea: el método CBR, `iCBR_Method`, que como hemos visto en el Capítulo 4 descompone la tarea en cuatro subtareas[1]: recuperar, adaptar, revisar y aprender.

Durante la fase de diseño de la aplicación, el diseñador debe elegir qué método se utilizará para resolver cada tarea concreta (Figura 5-2). Los siguientes apartados describen los métodos de la biblioteca de CBROnto, es decir, las alternativas que se ofrecen al diseñador de una aplicación CBR, organizados en torno a las cuatro subtareas que se derivan de la aplicación del método CBR: recuperación (Apartado 3), adaptación (Apartado 4), revisión (Apartado 5) y aprendizaje (Apartado 6).

3. Recuperación

Cualquier método de recuperación combina algún procedimiento para valorar el grado de similitud entre dos casos –consulta y objetivo– y algún procedimiento para buscar el caso más similar a la consulta. La investigación en el primer aspecto se ha centrado en desarrollar procedimientos eficientes y precisos para valorar la similitud semántica de un caso respecto a una consulta. Respecto al procedimiento de búsqueda la investigación se refiere a limitar o acotar la búsqueda para poder encontrar el mejor caso de forma más eficiente sin recorrer la base de casos completa pero sin reducir la calidad del resultado. Como hemos descrito en el Capítulo 2, la aproximación computacional pura, que consiste en llevar a cabo una búsqueda exhaustiva en la base de casos, aunque es simple y está muy extendida, puede resultar muy ineficiente, sobre todo para bases de casos grandes. La solución pasa por usar arquitecturas paralelas que se apoyan en un aumento de los recursos HW utilizados, o estrategias de búsqueda basadas en estructuras de organización que evitan tener que examinar todos los casos.

En el Capítulo 4 (Apartado 3.2.1.1) se describió el lenguaje de representación de métodos de CBROnto. Todos los métodos de recuperación comparten:

- La misma competencia –representada por el individuo canónico `iRetrieve_Task`.

[1] Realmente, como veremos en el Capítulo 6, el diseñador puede eliminar alguna de estas subtareas.

- La salida –representada por el individuo `iRetrieval_Output`– que incluye el atributo `retrieved_cases` con la lista de casos recuperados ordenados por su valor de similitud con la consulta.

Los siguientes apartados describen los métodos de la biblioteca de CBROnto (que hemos introducido en el Capítulo 4) y que resuelven la tarea de recuperación de casos: cómputo de similitud numérica (Apartado 3.1), clasificación de conceptos y reconocimiento de instancias, sobre una estructura conceptual genérica o sobre el retículo de conceptos formales resultante de aplicar el AFC a la base de casos (Apartado 3.2), criterios de relevancia (Apartado 3.3) y términos de similitud (Apartado 3.4). Para cada método se describe su comportamiento y sus componentes –según el lenguaje de representación de métodos. En concreto hacemos hincapié en los requisitos del método y las subtareas que genera. El lector interesado puede completar la descripción de los apartados siguientes con la formalización de los métodos que se incluye en el Apéndice B y el listado textual de métodos que se incluye en el Apéndice C.

3.1 Recuperación por cómputo de similitud

Este tipo de recuperación se caracteriza porque la similitud entre la consulta y los casos se valora usando una medida que se computa en tiempo de ejecución, es decir, representa una aproximación computacional (vs. representacional) al CBR. Aunque muchas métricas de similitud utilizadas en los sistemas CBR asumen que los casos se representan como colecciones de pares atributo-valor, en el Capítulo 2 hemos visto que las representaciones estructuradas, como las que nosotros planteamos, requieren el uso de medidas más complejas. En particular, nos centramos en las métricas de similitud que computan valores numéricos.

Nuestra propuesta se relaciona con la aproximación presentada en [Bergmann&Stahl98], descrita en el Capítulo 2, ya que divide el cómputo de la similitud en dos componentes que hemos llamado *similitud por posición* y *similitud por contenidos* y que se corresponden con la similitud *inter-class e intra-class*, respectivamente. Así, para computar la similitud entre dos objetos (individuos) se valora su similitud por posición, su similitud por contenidos y ambos resultados se combinan utilizando una cierta función de combinación:

- La componente de similitud por posición depende de la posición relativa de los dos individuos en la jerarquía conceptual del dominio, es decir, los conceptos a los que pertenecen.
- La componente de similitud por contenidos depende de los atributos o relaciones de los individuos comparados con otros individuos, que a su vez pueden ser estructurados o simples. Es decir, considera los atributos compartidos por los objetos comparados y la similitud entre sus rellenos. Para el cómputo de esta componente:
  - Si los individuos son estructurados, es decir, se relacionan con otros individuos mediante atributos relacionales, la similitud por contenidos se corresponde con lo que típicamente se llama similitud global. Se computa con una función que calcula recursivamente la similitud entre los rellenos de los atributos comunes y combina los resultados de similitud obtenidos. En el Capítulo 2 hemos descrito ejemplos clásicos de funciones de similitud global –media aritmética, euclidea, ...
  - Si los individuos son simples, es decir, no tienen atributos, no tiene sentido aplicar una función global de combinación. Por tanto, la similitud por contenidos se corresponde con la similitud local. Las funciones de similitud local son, en general, específicas de cada tipo (concepto simple). Puede ser una referencia a una tabla, a

una función definida por el usuario o a funciones predefinidas, por ejemplo para tipos numéricos o simbólicos (distancia, igualdad, etc.).

- Los dos resultados anteriores se combinan usando una función de combinación como la media ponderada o simplemente el producto normalizado entre los valores como en [Bergmann&Stahl98]. En nuestra propuesta la combinación se puede llevar a cabo usando a su vez cualquiera de las funciones de similitud global.

Nuestra aportación se basa en la representación explícita y declarativa de cada una de las componentes de las funciones de similitud utilizando terminología de CBROnto. El lenguaje de descripción de medidas de similitud de CBROnto permite definir nuevas medidas y asociarlas con ciertos conceptos del dominio. Estas medidas se utilizarán para comparar entre sí las instancias de dichos conceptos del dominio. Además, para aplicar una medida de similitud se tendrá en cuenta el tipo de índice asociado con el concepto actual (según describimos en el Capítulo 4 -Apartado 5.2.1). El tipo de índice define la forma de medir la similitud entre los individuos en función de los términos que contribuyen, es decir, que se puede medir la similitud en base a una parte de los atributos de un objeto ignorando el resto.

Los siguientes pasos esquematizan el *método de recuperación por cómputo de similitud*:

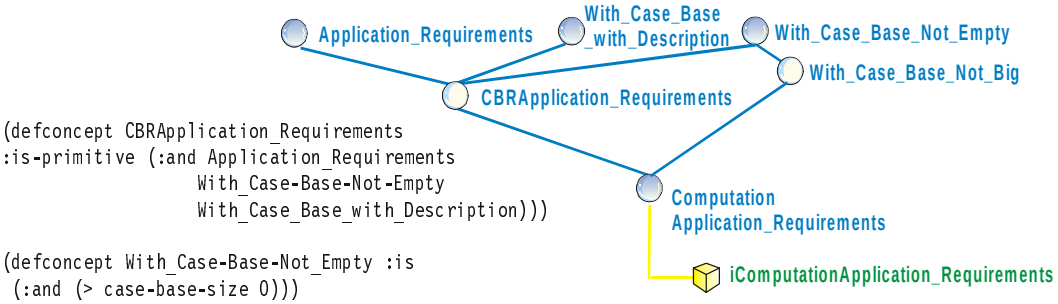
1. Se obtiene un conjunto inicial de casos candidatos.
2. Se compara el individuo consulta q con cada uno de los casos candidatos usando el siguiente método general:
    a. Para comparar dos individuos i_1 e i_2 cualesquiera se obtiene el concepto común más específico del que ambos son instancia Ci. Se accede a la medida de similitud asociada a este concepto Ci[2], que indica el modo de computar la similitud por contenidos, por posición y cómo combinarlas. Se computan los valores de similitud teniendo en cuenta únicamente las contribuciones de los atributos adecuados según el tipo de índice asociado al concepto Ci.
    b. Si i_1 e i_2 son individuos estructurados, la función global de similitud por contenidos calcula recursivamente la similitud entre los rellenos de sus atributos comunes, accediendo a su vez a la medida de similitud asociada al concepto más específico que contenga a ambos rellenos, y combina los resultados de similitud obtenidos.
3. Se puede utilizar un umbral para seleccionar los casos con un valor de similitud con la consulta mayor.

Los apartados siguientes describen los requisitos del método, las subtareas que genera y los métodos que las resuelven, y cómo se formalizan en CBROnto las funciones de similitud y las medidas de similitud que las utilizan.

3.1.1 Requisitos del método

La relación de subsunción de las DLs permite que CBROnto defina una jerarquía de herencia entre los conceptos que representan los requisitos de aplicabilidad de los métodos. El método de recuperación por cómputo de similitud hereda los requisitos de aplicabilidad del método CBR –que describimos en el Capítulo 4, Apartado 3.3.1– y especializa las restricciones sobre la base de casos, añadiendo un requisito de aplicabilidad adicional, por motivos de eficiencia, para evitar la aplicación de este método en bases de casos grandes.

[2] Realmente el concepto Ci no es único en general. El Apartado 3.1.3.1 describe el mecanismos de acceso a las medidas de similitud.




```
(defconcept CBRApplication_Requirements
:is-primitive (:and Application_Requirements
                    With_Case-Base-Not-Empty
                    With_Case_Base_with_Description)))

(defconcept With_Case-Base-Not_Empty :is
 (:and (> case-base-size 0)))

(defconcept With_Case_Base_Not_Big :is
 (:and (> case-base-size 0) (< case-base-size 50)))

(defconcept ComputationApplication_Requirements :is-primitive
(:and CBRApplication_Requirements With_Case_Base_Not_Big))
```

Figura 5-3. *Requisitos de aplicabilidad del método de recuperación por cómputo de similitud*

La Figura 5-3 muestra la relación de subsunción entre los conceptos que representan los requisitos de aplicabilidad de los métodos de recuperación por computación y el método CBR. El concepto `ComputationApplication_Requirements` hereda las características del concepto `CBRApplication_Requirements`, es decir, requiere una base de casos con descripción que contiene al menos un caso. El concepto `ComputationApplication_Requirements` restringe las características de la base de casos, especializando el concepto `With_Case_Base_Not_Empty` a `With_Case_Base_Not_Big`.

Este tipo de razonamiento sobre los conceptos de requisitos es el mecanismo que utilizamos para determinar la aplicabilidad de un método con respecto a un cierto contexto. Como se describe en el Capítulo 6 para determinar si un método es aplicable se construye un concepto que representa al contexto actual y se comprueba si se clasifica (automáticamente por sus características) como una especialización del concepto de requisitos de aplicabilidad del método. Si es así el método es aplicable en el contexto actual y si no, el propio concepto de requisitos de aplicabilidad permite explicar qué conocimiento le falta al contexto para poder aplicar el método.

La Figura 5-4 representa los requisitos de entrada –representados por el individuo `iComputation_Inputs`– del método de recuperación por cómputo de similitud (que está representado por el individuo canónico `iRetrieve_computational_method`):

- Los requisitos paramétricos son la especificación de la consulta (query) y de la base de casos (casebase) en la que se va a recuperar. Estos requisitos los hereda del método CBR. La base de casos se debe entender como uno de los tipos de casos definidos (subconceptos de `CASE`). Para buscar en todos los tipos de casos la base de casos especificada se referirá al concepto `CASE`. Los valores concretos que rellenan los atributos `query` y `casebase` del individuo `iComputationParameter`, es decir, la base de casos y la consulta con las que trabajará el método, se extraen de la descripción del individuo que representa el contexto actual (se describe en el Capítulo 6).
- Los requisitos de conocimiento especifican la conveniencia de definir alguna medida de similitud asociada al concepto tipo de caso que representa la base de casos.

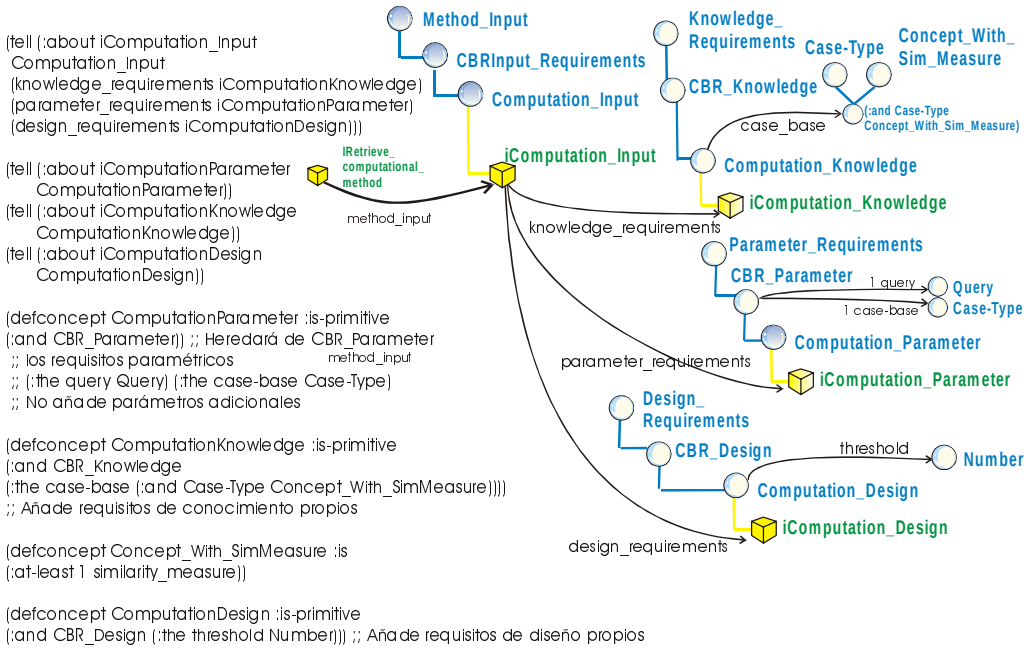


Figura 5-4. *Requisitos de entrada del método de recuperación por cómputo de similitud*

- Como requisitos de diseño, el diseñador puede incluir el umbral de similitud a partir del cual un individuo se considera adecuado. Si no se especifica umbral se devuelven todos los casos (que estarán ordenados en orden decreciente según el valor de similitud con la consulta), es decir, se considera umbral cero.

Puesto que la tarea de recuperación es la primera de la secuencia, ningún método de recuperación tendrá requisitos de secuencia.

3.1.2 Resolución de las subtareas

El método de recuperación por cómputo de similitud es un método de descomposición —es decir, instancia de `Decomposition_Method`— que, dado un individuo consulta, divide la tarea de recuperación en tres subtareas: obtener los casos de la base de casos –`ObtainCases_Task` (instancia canónica `iobtain_cases`)–, comparar la consulta con cada uno de ellos utilizando una cierta medida de similitud y ordenarlos de mayor a menor valor de similitud –`Assess-Sim_Task` (`icompute_similarity`)– y teniendo en cuenta los valores de similitud seleccionar el/los casos a recuperar –`Select_Task` (`iselect_best`).

Cada una de estas subtareas se resuelve a su vez por otros métodos que sólo añaden a los requisitos que se heredan del nivel superior los requisitos de secuencia que garantizan que se pueden secuencializar con los métodos cuya salida sea compatible.

- La tarea `iobtain_cases` obtiene el conjunto inicial de casos candidatos al que se aplicarán las siguientes subtareas. Esta tarea está asociada (mediante el enlace `task_method`) con el único método que la resuelve (`iobtain_case_base`) que obtiene todas las instancias de la base de casos que recibe especificada en el parámetro `case-base` (heredado). El método genera como salida un individuo (`iob-`

tain_cases_output) con un atributo caseList que lo relaciona con el conjunto de casos resultantes.

- La tarea icompute_similarity tiene asociado un método (inumeric_simcomputation_method) que se encarga de computar la similitud entre la consulta (parámetro query) y cada uno de los casos del conjunto anterior (requisito de secuencia caseList). El método genera como salida un individuo (icompute_similarity_output) con un atributo caseList que lo relaciona con el conjunto de casos de salida. Este atributo existe porque es un requisito de secuencia de los métodos de selección que se resuelven a continuación en la secuencia. Sin embargo, en el caso concreto del método de cómputo de similitud devuelve también un atributo simValues con una lista de pares (Ci, SVi), donde Ci es el nombre de un caso y SVi el valor de similitud entre el caso Ci y la consulta. La lista está ordenada de mayor a menor valor de similitud.
- La tarea de selección (iselect_best) se encarga de elegir el caso mejor teniendo en cuenta la información de similitud obtenida por el método anterior y la información del resultado del caso (si existe). Existen tres métodos cuya competencia es adecuada para resolver esta tarea:
  - iuserselectcase_method, ofrece la lista de opciones y delega al usuario la selección del caso mejor. Incluye como requisito de secuencia la existencia del atributo caseList con la lista de casos candidatos.
  - iselectallcases_method, selecciona todos los casos con un valor de similitud superior al valor umbral (requisito de diseño heredado threshold). Como este método se puede utilizar también para seleccionar casos en métodos en los que el cómputo de similitud no es numérico, sólo incluye como requisito de secuencia la existencia del atributo caseList (y selecciona todos sus casos sin tener en cuenta el umbral), y no el atributo simValues que sólo se devuelve si hay cómputo de similitud numérica. Si existe el atributo simValues lo utiliza para seleccionar todos los casos con similitud mayor al umbral dado.
  - iselectmaxcase_method, selecciona el primer caso (el de mayor valor de similitud) de la lista que recibe. Si el caso más similar incluye un resultado de fallo, entonces se seleccionan también los siguientes hasta incluir alguno con resultado de éxito. Igual que el método anterior incluye como requisito de secuencia la existencia del atributo caseList aunque si existe el atributo simValues lo utiliza.

  Por estar los tres métodos asociados a la última subtarea de la secuencia de tareas derivadas por los métodos de selección, generan como salida el mismo individuo iRetrieval_Output que devolverán como salida los métodos de recuperación.

Obviamente, la tarea más importante de las tres es la que se encarga del cómputo de similitud (icompute_similarity). El método que se encarga de su resolución involucra varias subtareas (que no se representan explícitamente ya que lo hemos considerado un método de resolución) que están implícitas en el código Lisp de la función que implementa la especificación operacional del método. En concreto, el método recupera la medida de similitud que debe utilizar para comparar la consulta y los casos, accede a cada una de sus componentes (posición y contenidos) y los combina usando la función de combinación.

Los siguientes apartados describen cómo se formalizan las medidas de similitud, cómo se asocian a los conceptos del dominio y el mecanismo que utiliza el método de cómputo de

similitud para acceder (recursivamente) a la medida de similitud adecuada según la posición de los individuos comparados.

3.1.3 Formalización de las medidas de similitud en CBROnto

CBROnto representa explícita y declarativamente ciertas medidas y funciones de similitud predefinidas y ofrece un lenguaje para definir nuevas medidas de similitud específicas del dominio o de la aplicación concreta.

En CBROnto cada medida de similitud se representa como una instancia del concepto primitivo `SimilarityMeasure`:

```
(defconcept similarityMeasure
 :is-primitive (:and CBROnto-concept
  (:all contents contentssimilarityFunction) (:at-most 1 contents)
  (:all position positionsimilarityFunction) (:at-most 1 position)
  (:all combination globalsimilarityFunction)(:at-most 1 combination)
  (:all multivalued globalsimilarityFunction) (:at-most 1 multivalued)
  (:at-most 1 weigth))
  :annotations ((documentation "the class of all the individuals representing a simi-
larity measure to compare instances of a concept.")))
```

La relación `currentMeasure` permite *anotar* un concepto con la medida de similitud que será utilizada para computar la similitud entre dos instancias cualesquiera de ese concepto.

Como hemos descrito, el cómputo de la similitud se divide en dos componentes, posición y contenidos, que se combinan. Por tanto, en cada individuo medida de similitud –instancia de `SimilarityMeasure`– se especifica cómo computar las componentes de similitud por contenidos y por posición y cómo combinar sus resultados. Los atributos `contents`, `position` y `combination` relacionan la medida de similitud con las funciones de similitud –instancias del concepto `similarityFunction`– que se utilizarán para calcular las componentes de contenidos, posición y combinar sus resultados, respectivamente.

Cuando una medida de similitud está asociada (a través de la relación `currentMeasure`) con un concepto "simple", es decir cuyas instancias son individuos sin atributos relacionales[3], la similitud por contenidos se computará usando una *función de similitud local* no recursiva que computa un valor de similitud. Cuando la medida de similitud se asocie a conceptos estructurados la componente de similitud por contenidos se computa usando una *función de similitud global* que indica cómo combinar los resultados de similitud obtenidos al comparar los rellenos de cada uno de los atributos comunes que pertenezcan al tipo de índice aplicado. En ambos casos la similitud por posición depende de la situación en la jerarquía del concepto más específico que contiene a los dos objetos comparados. Además, una medida de similitud asociada a un concepto tendrá en cuenta el tipo de índice asociado a dicho concepto, cuando haya alguno, o el tipo de índice semántico que se usa por defecto.

Para comparar la similitud de los rellenos de los atributos *multivalorados* habrá que computar la similitud entre dos conjuntos de valores. Se computa la similitud entre cada elemento de un conjunto y todos los del otro conjunto y, utilizando alguna función se combinan todos los valores de similitud obtenidos. El atributo `multivalued` de la medida de similitud permite indicar la función de combinación que se utilizará (cualquiera de las funciones globales). Si no se especifica esta componente, el comportamiento por defecto utiliza la función `máximo`,

[3] O que, aunque tengan atributos, no se quieran tener en cuenta.

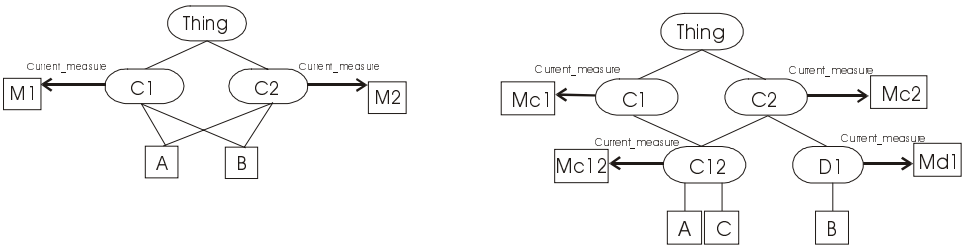


Figura 5-5. *Medidas de Similitud*

de forma que la similitud entre los dos conjuntos de valores se corresponde con la máxima similitud que exista entre dos elementos cualesquiera, uno de cada conjunto.

El atributo `weigth` de la medida de similitud permite al diseñador asignar un peso a una medida de similitud para que el mecanismo de acceso, que describimos a continuación, pueda resolver conflictos cuando haya varias medidas aplicables.

3.1.3.1 Mecanismo de acceso a las medidas de similitud

El primer paso para computar la similitud entre dos individuos cualesquiera es la recuperación de la medida de similitud que se utilizará. El uso de una u otra medida depende de la elección particular del diseñador de la aplicación que induce una asociación de una cierta medida de similitud con un concepto del dominio o de CBROnto.

El método básico consiste en acceder al concepto más específico que contiene a los dos individuos comparados, es decir, del que ambos son instancia, y obtener su medida de similitud asociada a través de la relación `current_measure`. Este proceso utilizará la jerarquía de subsunción para recorrer los niveles conceptuales, ya que no podemos suponer que todos los conceptos estén anotados con una medida de similitud que determine cómo comparar sus instancias. Si un concepto no tiene una medida de similitud asociada podemos subir en la jerarquía hasta encontrar el concepto más específico que contiene a los individuos y que tiene una medida de similitud vinculada. Una vez obtenida, la instancia medida de similitud nos proporciona la información sobre qué función de similitud por posición, qué función de similitud por contenidos y qué función de combinación se va a utilizar.

Un problema de este mecanismo se debe al hecho de, como se muestra en la Figura 5-5, el concepto más específico que contiene a dos individuos dados no es único en general, lo que podría llevar a obtener varias medidas de similitud aplicables. Esta situación puede ser considerada una decisión de diseño, ya que es el diseñador de una aplicación el que inducirá los vínculos entre los conceptos y las medidas de similitud. Además, el mecanismo de anotaciones de las medidas de similitud en los conceptos, permite que el diseñador asocie más de una medida de similitud para un mismo concepto, indicando formas alternativas de valorar la similitud entre las instancias de ese concepto.

Sea cuál sea la causa, cuando existen varias medidas aplicables se podría seleccionar una de las medidas al azar, dejar que el usuario final decida cuál utilizar, o incluir criterios que permitan al sistema elegir una de ellas en función, por ejemplo, del peso asociado a las medidas de similitud. Nuestra elección es permitir que el diseñador asigne pesos a las medidas de similitud de forma que, cuando hay varias medidas alternativas se seleccionará la de mayor peso. Si dos medidas tienen el mismo peso (en particular ninguno), el sistema utilizará un método alternativo para acceder a la medida de similitud, basado en el rango de la relación de la que los dos individuos comparados son relleno, que describimos a continuación, y si continúa el conflicto, se planteará una consulta al usuario final del sistema CBR.

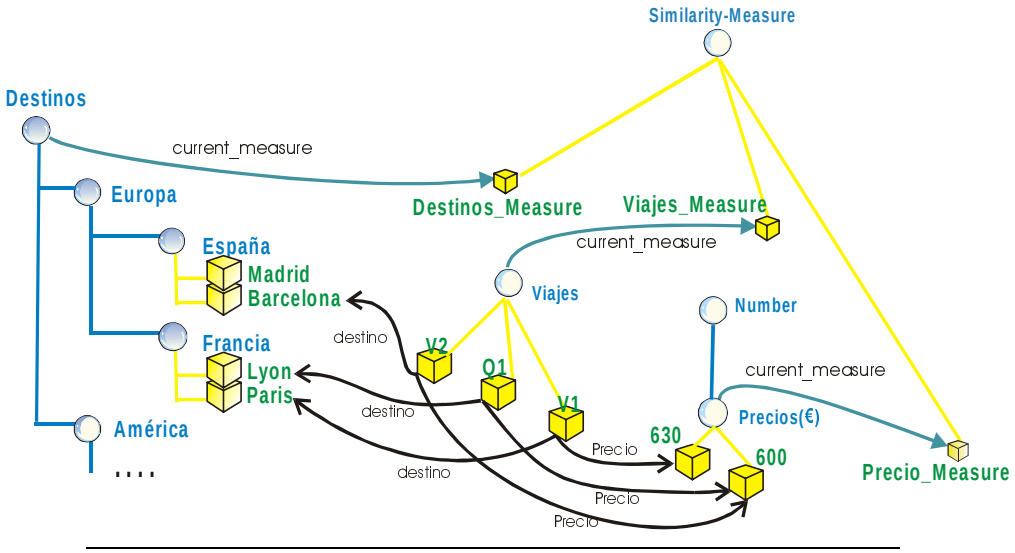


Figura 5-6. *Ejemplo de cómputo de similitud*

En la situación representada en la Figura 5-5 (izquierda) existe un conflicto ya que los individuos A y B son instancia de los conceptos C1 y C2, con el mismo nivel de especificidad, y ambos conceptos tienen medidas de similitud asociadas. En la Figura 5-5 (derecha) para medir la similitud entre los individuos A y C usaríamos la medida de similitud representada por Mc12 (instancia de `Similarity_Measure`). Si no existiera el vínculo entre C12 y Mc12 se produce un conflicto similar al anterior en el que dos medidas de similitud (Mc1 y Mc2) son aplicables. Para medir la similitud entre A y B no existe conflicto y se usaría Mc2 que es la medida asociada a C2 que es el concepto más específico del que ambos son instancia.

Además del mecanismo anterior, existe una segunda forma de obtener la medida de similitud adecuada para comparar dos individuos y que resuelve en cierta medida los problemas anteriores. En vez de obtener la medida de similitud asociada al concepto del que los individuos comparados son instancia, este método consiste en obtener la medida de similitud a partir del concepto que representa el rango de la relación (o atributo relacional) de la que ambos individuos son rellenos. Esto es correcto, ya que el mecanismo de chequeo de consistencia garantiza que ambos individuos serán instancias de este concepto. Además, este método es especialmente adecuado para comparar números en nuestro sistema de representación. Por ejemplo, supongamos que queremos comparar las instancias `Q1` y `V1` del concepto `Viajes` de la Figura 5-6 y supongamos también que la medida de similitud asociada (`Viajes_Measure`) indica que la similitud por contenidos resulta de la media aritmética de los valores de similitud entre los rellenos de sus dos atributos relacionales `destino` y `precio`:

```
Content-Similarity (Q1, V1) = (Sim(Lyon, París) + Sim(600, 630))/2
```

Para computar la componente de similitud `Sim(Lyon, París)`, el sistema de DL obtiene los conceptos a los que pertenecen los individuos `París` y `Lyon`, y recupera la medida de similitud `Destinos_Measure` asociada con `Destinos` que es el concepto más específico del que ambos individuos son instancia y que tiene asignada una medida de similitud, ya que ni `Francia` ni `Europa` tienen medidas asociadas.

Sin embargo, para computar la componente de similitud `Sim(600, 630)`, surge un problema –tecnicismo de implementación– derivado del uso de individuos que representan valores numéricos. Al utilizar el método del concepto más específico del que ambos indivi-

duos (600 y 630) son instancias, LOOM no puede inferir que los números representan *precios* y por tanto, no los reconoce como instancias del concepto Precios. Sin embargo los reconoce como instancias de los conceptos siguientes: (|C|INTEGER |C|RATIONAL-NUMBER |C|NUMBER |C|LOOM::QUANTITY |C|CONSTANT |C|NON-LOOM-THING |C|THING).

En el ejemplo nos interesa distinguir que los números 600 y 630 que estamos comparando son precios en Euros, y no otro tipo de números. En este caso se obtiene la medida de similitud adecuada usando un método alternativo que tiene en cuenta el rango de la relación considerada, el concepto Price para obtener la medida de similitud asociada. En este ejemplo el rango de la relación precio es el concepto Precios que nos da acceso a la medida Precio_Measure.

En general, utilizaremos la medida de similitud asociada al concepto más específico que resulte de comparar el concepto más específico que contiene a los objetos comparados y el rango del atributo relacional actual. Por ejemplo, si el concepto Francia tuviese asociada una medida de similitud deberíamos utilizarla para comparar los individuos París y Lyon, aunque el rango de la relación destino sea el concepto Destinos.

La estructura de las instancias Precio_Measure, Viajes_Measure y Destino_Measure se describe en el Apartado 3.1.4 dedicado a la formalización de las funciones de similitud.

3.1.3.2 Medidas de similitud específicas de un tipo de consulta

Antes de aplicar el mecanismo de acceso a las medidas de similitud que hemos descrito en el apartado anterior existe una excepción que se aplica sólo si estamos comparando un individuo consulta q con un individuo caso c, y el tipo de consulta al que pertenece el individuo q tiene asociado una medida de similitud específica. Esto representa una excepción al método general ya que el caso c en general no tiene por qué ser instancia del concepto tipo de consulta.

Esta excepción al método general permite que el diseñador prevea el uso de distintas medidas de similitud dependiendo de las características de la consulta, y no únicamente del tipo de caso. Como pauta de diseño la medida de similitud se asocia con uno de los concepto tipo de caso si se quiere el mismo comportamiento para todos los casos del tipo; y se asocia con el tipo de consulta para variar el comportamiento según la consulta que el usuario final plantee al sistema.

En el ejemplo de la agencia de viajes, supongamos que todas las consultas deben especificar uno o varios destinos, y siempre es obligatorio especificar la estación o época del año en la que se quiere hacer el viaje. El diseñador puede definir como tipo de consulta el siguiente concepto y anotar las medidas de similitud que se aplicarán para valorar la similitud entre una consulta de este tipo y un caso:

```
(defconcept varios-destinos :is (:and Query-Type
          (:at-least 2 destino) (:all destino Destinos)
          (:at-least 1 estación) (:all estación Estaciones)))
(tell (:about varios-destinos (current-measure Destinos-Alternativos)
                        (current-measure Destinos-Simultaneos)))
```

En concreto asociaremos dos medidas de similitud sin especificar el peso para que sea el usuario final el que decida cómo interpretar la consulta. La primera medida interpreta los destinos como alternativos, para recuperar casos cuyo destino sea *alguno* de los pedidos. La medida Destinos-Simultaneos, se utilizará para recuperar viajes que cubran la mayor cantidad de los destinos pedidos. Observamos que los casos de la base de casos no pertenecen a estos conceptos, por lo usando el método general no tendríamos acceso a estas medidas.

Describimos a continuación cómo se representan en CBROnto las funciones de similitud que forman parte de las medidas de similitud que hemos descrito en este apartado.

3.1.4 Formalización de las funciones de similitud

En CBROnto se representan ciertas funciones que, como hemos descrito, se utilizan para computar los valores de similitud durante la recuperación de casos. Cada función de similitud se representa como una instancia del concepto `SimilarityFunction` y se describe mediante el nombre de una función Lisp que implementa la función de similitud así como valores para sus parámetros:

```
 (defconcept similarityFunction:is-primitive
    (:and (:exactly 1 function-name) (:all parameters function-parameter))
  :annotations ((documentation"the class of all the individuals representing
     a similarity function to compute a similarity value.")))
(defconcept localsimilarityFunction :is-primitive similarityFunction)
(defconcept globalsimilarityFunction :is-primitive similarityFunction)
(defconcept positionsimilarityFunction :is-primitive similarityFunction)
```

Distinguimos entre tres tipos de funciones de similitud que se representan mediante subconceptos de `similarityFunction`:

- Las instancias de `localsimilarityFunction` representan funciones locales que son las que típicamente se asocian a los conceptos simples o "tipos básicos". Representan las funciones definidas para comparar dos individuos sin atributos.
- Las instancias de `globalsimilarityFunction` representan funciones globales que indican cómo computar la similitud entre dos objetos estructurados. Son funciones que calculan recursivamente la similitud entre los rellenos de los atributos (simples o relacionales) y combinan los resultados obtenidos.
- Las instancias de `positionsimilarityFunction` representan funciones de similitud por posición que indican cómo calcular la similitud entre dos individuos debida a la posición en la jerarquía del concepto más específico que los contenga a ambos.

Los siguientes apartados describen las funciones de similitud predefinidas en CBROnto cada una de las cuales se representa como un individuo que es instancia de alguno de estos conceptos. Cuando el método de recuperación por cómputo de similitud accede a una instancia función de similitud utiliza una sencilla función de orden superior que aplica la función usando su nombre y como parámetro la lista que resulta de concatenar los dos individuos con los parámetros representados en la instancia: `(apply nombre-funcion (i1 i2 parámetros))`.

3.1.4.1 Funciones de similitud local

Las funciones de similitud local se usan típicamente para comparar dos individuos que son instancias de un concepto simple, es decir, sin atributos. Existen algunas que son aplicables a cualquier tipo y otras que son específicas de un tipo concreto. La tabla siguiente resume las funciones de similitud local genéricas que están predefinidas en CBROnto:

Nombre	*Descripción*	*Aplicable a*		
		Símbolos[4]	*Números*	*Cadenas*
Igualdad	$sim(a,b) = \begin{cases} 0, \text{si a} \neq \text{b} \\ 1, \text{si a} = \text{b} \end{cases}$	☑	☑	☑
Tabla	consulta una tabla de similitudes (descrita por el diseñador) en la que se da explícitamente el valor entre cada dos elementos	☑	☑	☑
Llamada a función	aplica una función específica del dominio programada por el diseñador	☑	☑	☑
Intervalo	$sim(a,b) = 1 - \frac{\lvert a-b \rvert}{l}$ donde a y b son valores de un intervalo numérico de longitud l		☑	
Salto	$sim(a,b) = \begin{cases} 1, si \lvert a-b \rvert \leq d \\ 0, e.o.c \end{cases}$ donde d es una distancia dada		☑	
Perfección menor (mayor)	$sim(a,b) = \begin{cases} 1, si \lvert a-b \rvert \leq d \\ \text{intervalo(a, b)}, e.o.c \end{cases}$		☑	
Encaje de cadenas (con o sin distinción de mayúsculas)	$sim(a,b) = \begin{cases} 1, & \text{Si a y b son cadenas iguales} \\ 0, e.o.c \end{cases}$			☑
Chequeo de deletreo	$sim(a,b) =$ letras comunes entre a y b sin tener en cuenta la posición de las mismas			☑
Cuenta de palabras	$sim(a,b) =$ compara frases contando el número de palabras iguales			☑

3.1.4.2 Funciones de similitud global

Las funciones globales se encargan de combinar los resultados de similitud local. Se utilizan para la componente de combinación de las medidas de similitud y para generar las llamadas recursivas, en la componente de similitud por contenidos de las medidas asociadas a los conceptos estructurados. CBROnto representa todas las funciones globales que hemos descrito en el Capítulo 2: bloques, bloques ponderada, Minkowski normal y ponderada, máximo, mínimo y media aritmética y ponderada. La formalización de estas funciones, igual que para las locales y posición, se puede consultar en el Apéndice B.

[4] En general, las funciones aplicables a símbolos se pueden aplicar a las instancias de cualquier concepto del dominio.

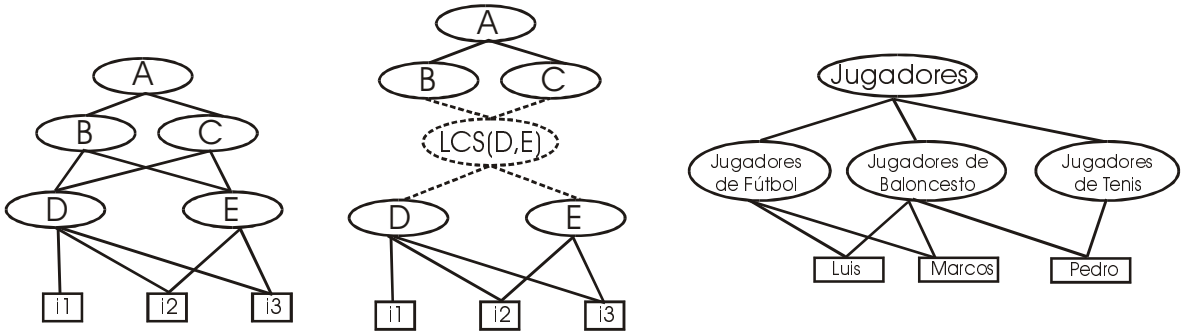


Figura 5-7. *En el espacio de conceptos definidos el concepto común más específico entre dos conceptos no es único*

3.1.4.3 Funciones de similitud por posición

Las funciones de similitud por posición se utilizan para determinar la componente de similitud que depende de la posición que ocupan los dos individuos comparados en la taxonomía conceptual del dominio, es decir, depende de los conceptos a los que pertenecen los individuos. La componente de similitud por posición no tiene en cuenta explícitamente las relaciones de los individuos, es decir, los valores de sus atributos –que se consideran en la componente de similitud por contenidos. Sin embargo, las relaciones intervienen de forma implícita ya que la pertenencia o no de un individuo a un concepto depende también de cómo se relacione con otros individuos[5].

Todas las funciones de similitud por posición de CBROnto se basan en computar la posición que ocupa en la jerarquía el concepto más específico que contiene a los dos individuos comparados, que es el que representa la estructura común a ambos. Como vimos en el Capítulo 3, el concepto común más específico entre dos conceptos o LCS (*Least Common Subsummer*) es el concepto más específico que subsume a ambos conceptos. El LCS es único en el espacio abstracto de conceptos potencialmente definibles utilizando el lenguaje de descripción de conceptos. Pero si utilizamos únicamente los conceptos que actualmente están definidos en una base de conocimiento, el concepto común más específico entre dos conceptos no es único en general (ver Figura 5-7). Lo mismo ocurre desde el punto de vista de las instancias: el concepto más específico de la base de conocimiento del que dos individuos son instancia es único en el espacio abstracto de conceptos potencialmente definibles utilizando el lenguaje de descripción de conceptos. Pero si no queremos añadir conceptos adicionales, sino sólo utilizar los conceptos del modelo del dominio, no podemos hablar del concepto más específico sino del conjunto de conceptos más específicos que contienen a los dos individuos. En el ejemplo de la Figura 5-7 el conjunto de conceptos más específicos que contienen a los individuos i2 e i3 es {D, E}.

En general, para comparar la posición de los individuos en la jerarquía tendremos en cuenta todos los conceptos de los que los individuos son instancia en vez del concepto más específico que puede no estar definido en la base de conocimiento actual.

Relacionada con nuestra aproximación, que asocia explícitamente las medidas de similitud con los conceptos del dominio, existiría otra alternativa que se basa en computar la similitud

[5] Ya que en la definición del concepto se caracterizan las relaciones que las instancias de este concepto tendrán con otros individuos.

por posición de dos individuos utilizando la posición del único concepto, al que ambas instancias pertenecen, cuya medida de similitud estamos utilizando. De este modo la similitud se computa según la profundidad de ese concepto y no de otros a los que también pueden pertenecer las instancias. Esta aproximación es justificable ya que estamos utilizando una cierta instancia medida de similitud asociada a un concepto y es la posición de ese concepto la que recibe una importancia distinguida. Por ejemplo en la Figura 5-7 (derecha) los individuos "Luis" y "Marcos" son instancia de los conceptos "Jugadores de Baloncesto" y "Jugadores de Fútbol". Supongamos que cada uno de dichos conceptos tuviera asociada una medida de similitud de forma que, como ambas son aplicables, el diseñador puede indicar cuál utilizar (asociando a la elegida un peso mayor) o dejar la elección al usuario final. En cualquier caso, para comparar a "Luis" y a "Marcos" como jugadores de baloncesto la similitud tiene en cuenta la posición del concepto "Jugadores de Baloncesto" y no del concepto "Jugadores de Fútbol" al que ambos también pertenecen. Así la similitud por posición entre "Pedro" y "Marcos", y entre "Luis" y "Marcos" es la misma. Recíprocamente, si la medida de similitud elegida fuese la de "Jugadores de Fútbol" sería únicamente la posición de ese concepto la que intervendría en el cómputo de similitud.

Sin embargo, no hemos escogido esta aproximación ya que, aunque la comparación entre dos individuos se deba a la pertenencia a un concepto y por ello usemos su medida de similitud y no otra, el hecho de que los individuos compartan más características es relevante y hace que sean más similares. En el ejemplo la similitud por posición entre "Luis" y "Marcos" debería ser mayor que la de "Pedro" y "Marcos", ya que comparten más características (conceptos). Supongamos otro ejemplo en el que los conceptos "Jugadores de Baloncesto" y "Jugadores de Fútbol" no tuvieran asociada ninguna medida de similitud, y el concepto más específico que tiene asociada una medida de similitud fuese el concepto "Jugadores". Con la aproximación anterior usaríamos la posición de "Jugadores" para medir la similitud y no tendríamos en cuenta la pertenencia de estos individuos a otros conceptos. El resultado de similitud por posición entre "Luis" y "Marcos", entre "Marcos" y "Pedro" y entre "Luis" y "Pedro" sería el mismo ya que en todos los casos los dos individuos comparados son "Jugadores". La aproximación que nosotros hemos elegido, y que se usa en las funciones de similitud por posición que describimos a continuación, "Luis" y "Marcos" son los individuos más similares entre sí, y "Luis" y "Pedro", y "Marcos" y "Pedro" obtienen el mismo valor de similitud independientemente del concepto utilizado para obtener la medida de similitud. En nuestra aproximación, a la similitud de dos individuos contribuye la posición de todos los conceptos de los que los individuos son instancia.

Describimos a continuación las funciones de similitud por posición que están predefinidas en CBROnto. Su formalización se puede consultar en el Apéndice B.

Función de similitud constante

La función más sencilla que podemos definir es aquella que, independientemente de los individuos comparados, siempre devuelve un valor constante. Por ejemplo, llamamos *Fcons (i1,i2, c)* a una función de similitud que devuelve un valor constante c. Hemos incluido esta función para implementar la medida de similitud por posición descrita en [Bergmann&Stahl98] que propone anotar los nodos con un valor de similitud que es creciente con la profundidad del nodo en la jerarquía. En nuestro marco de similitud estas anotaciones se corresponden con medidas de similitud cuya componente de posición se calcule usando una función de similitud constante creciente con la profundidad de los conceptos. La correspondencia con nuestro marco se basa en sustituir cada valor con el que proponen anotar cada nodo, con una función de similitud por posición con dicho valor constante.

Función de similitud profundidad básica

Este segundo tipo de función de similitud computa los valores de similitud, en vez de anotarlos manualmente, en base a la relación existente entre la profundidad del concepto más específico de todos los que contienen a los dos individuos comparados y la profundidad máxima de la jerarquía.

$$fdeep\_basic(i1,i2) = \frac{\max(prof(LCS(i1,i2)))}{\max(prof(Ci)), Ci \in CN}$$

Donde:

- *CN* es el conjunto de conceptos de la base de conocimiento actual.
- *LCS(i1,i2)* es el conjunto de los conceptos (existentes en la base de conocimiento) más específicos que contienen a los dos individuos comparados.
- *Prof(Ci):* profundidad del concepto Ci que se computa como 1 + el número de enlaces de subsunción desde el elemento TOP hasta el concepto Ci.

Función de similitud profundidad

De forma similar a la anterior, esta función computa la similitud según la relación que exista entre la profundidad del concepto más específico que contiene a los dos individuos comparados y la del individuo más profundo de los dos.

$$fdeep(i1,i2) = \frac{\max(prof(LCS(i1,i2)))}{\max(prof(i1), prof(i2))}$$

Donde:

- *Prof(i):* profundidad del individuo i, es decir, número de enlaces de subsunción desde el elemento TOP hasta el individuo i. En este caso no es necesario sumar 1 porque las instancias están un nivel más abajo que los conceptos.

Función de similitud coseno

Se basa en la función de similitud entre conceptos definida en [González-Calero97] e inspirada en el modelo del espacio vectorial. Según este modelo, utilizado típicamente en los sistemas de recuperación de información, los documentos se representan mediante vectores de pesos de términos y la similitud entre documentos se calcula a través de la similitud entre los vectores que representan a dichos documentos. Entre las distintas funciones propuestas en la literatura, una que proporciona resultados satisfactorios y que es, al mismo tiempo, sencilla de calcular es la expresión que define la similitud entre dos vectores como el coseno del ángulo que forman [Salton&McGill83].

La función de similitud *coseno* se basa en representar cada concepto mediante un vector de propiedades y calcular la similitud entre dos conceptos como el resultado de aplicar una cierta función a los vectores que los representan. Cada concepto se representa mediante un vector con tantas componentes como conceptos pertenezcan a la base de conocimiento y cuyas componentes son 1 o 0, dependiendo de si el concepto representado es o no un subconcepto del concepto asociado a esa componente. En [González-Calero97] se propone una función de similitud entre dos instancias basada en el coseno del ángulo formado por los vectores que representan a los conceptos a los que pertenecen las instancias. Si definimos la similitud entre dos conceptos, c_i y c_j, como el coseno del ángulo que forman los vectores que los repre-

sentan, v_i y v_j, entonces la similitud vendrá dada por la expresión siguiente que toma valores entre 0 y 1, ya que todas las componentes de los vectores son positivas.

$$\text{sim}(c_i, c_j) := \frac{v_i \cdot v_j}{\|v_i\| \cdot \|v_j\|}$$

Al calcular la similitud entre dos conceptos, no construimos los vectores que los representan explícitamente, sino que obtenemos los conjuntos de superconceptos de cada uno de los conceptos a comparar y utilizamos dichos conjuntos para realizar los cálculos. La obtención de los superconceptos es una operación de complejidad lineal, dado que en la jerarquía conceptual hay enlaces explícitos entre cada concepto y sus superconceptos directos.

En [González-Calero97] se demuestra que esta función de similitud se computa mediante la expresión siguiente:

$$sim(i_1, i_2) = sim(t(i_1), t(i_2)) = \frac{\left| \left(\bigcup_{d_i \in t(i_1)} \text{super}(d_i, CN) \right) \cap \left(\bigcup_{d_i \in t(i_2)} \text{super}(d_i, CN) \right) \right|}{\sqrt{\left| \bigcup_{d_i \in t(i_1)} \text{super}(d_i, CN) \right|} \cdot \sqrt{\left| \bigcup_{d_i \in t(i_2)} \text{super}(d_i, CN) \right|}}$$

Donde:

- super(c, C) es el conjunto de superconceptos de c en C, siendo C un conjunto de conceptos dado y c un concepto que pertenece a dicho conjunto:

  $$\text{super}(c, C) := \{x \in C \mid c \prec x\}$$

- t(i) es el *tipo* o conjunto de conceptos pertenecientes a CN de los que i es instancia: $\forall i \in IN$, $\text{t}(i) := \{c \in CN \mid i \text{ es\_instancia\_de } c\}$, siendo IN el conjunto de instancias que se han definido en la base de conocimiento actual.

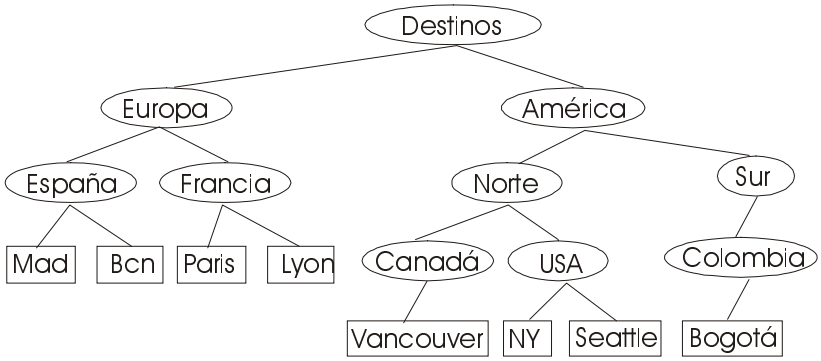
Función de similitud detalle

En esta función, la similitud por posición de dos individuos se define en función del número total de conceptos de los que los individuos son instancia. De esta forma, cuanto mayor sea el número de conceptos de los que son instancia, es decir, cuanto mayor sea el nivel de detalle o de información con el que se describen, mayor es la similitud.

$$\text{detalle}(i_1, i_2) = \text{detalle}(t(i_1), t(i_2)) = 1 - \frac{1}{2 \cdot \left| \left(\bigcup_{d_i \in t(i_1)} \text{super}(d_i, CN) \right) \cap \left(\bigcup_{d_i \in t(i_2)} \text{super}(d_i, CN) \right) \right|}$$

Ejemplo de aplicación de las funciones de similitud por posición

La Figura 5-8 muestra un ejemplo de aplicación de las funciones de posición descritas. Observando los resultados obtenidos podemos concluir que todas ellas comparten la propiedad de que dos individuos son mas similares cuanto menor sea la distancia que los separa en la taxonomía de conceptos. Sin embargo, los resultados varían ligeramente en el rango de valores obtenidos. Por ejemplo, las funciones de profundidad (`fdeep` y `fdeep_basic`) obtienen un valor de similitud 0 para aquellos individuos que sólo tengan un concepto común (concepto raíz), mientras que en el resto de las funciones el valor mínimo es mayor que 0 para reflejar



	fcons_c	fdeep_basic	fdeep	coseno	detalle
Sim(Mad, Bcn)	c	3/4	1	1	5/6
Sim(Mad, París)	c	1/2	1/2	2/3	3/4
Sim(NY, Seattle)	c	1	1	1	7/8
Sim(Seattle,Vanc)	c	3/4	3/4	3/4	5/6
Sim(Bcn, NY)	c	0	0	$1/_{l2}$	1/2
Sim(Mad,Bogotá)	c	0	0	$1/_{l2}$	½
Sim(Vanc,Bogotá)	c	1/2	1/2	1/2	3/4

Figura 5-8. *Ejemplo de aplicación de las funciones de similitud por posición*

que los individuos son instancia de al menos un concepto común. El menor valor obtenido por la función `detalle` es ½ por lo que esta función deberá ser elegida para una medida de similitud en la que la componente por posición tenga un peso importante. Además esta función sólo obtiene valor 1 cuando los individuos son iguales, al contrario que las funciones `coseno` y `profundidad`, que computan un valor de similitud 1 para dos instancias del mismo concepto, aunque no sea el mismo individuo. La función `profundidad básica` sólo asigna 1 a las instancias del mismo concepto si éste es el más profundo de la jerarquía.

Aunque estas pequeñas diferencias pueden ser tenidas en cuenta para un diseñador que cree una medida de similitud, el uso de una u otra no varía significativamente el resultado obtenido. Cualquiera de ellas puede resultar adecuada sobre todo si se ajusta su importancia a través de una función de combinación ponderada que asigne más o menos peso a la componente de similitud por posición.

3.1.4.4 Ejemplo de aplicación del método de cómputo de similitud

El método que se encarga de resolver la subtarea de valoración de la similitud (`inumeric_simcomputation_method`) computará la similitud entre la consulta y cada uno de los casos del conjunto `caseList` obtenido en la subtarea anterior (obtención de casos).

Supongamos que el método tiene que comparar la consulta `Q1` con los viajes `V1` y `V2` de la Figura 5-9, donde la consulta se refiere a un viaje con destino `Lyon` y de precio 600€, `V1` representa un viaje a `París` por 630€, y el viaje `V2` un viaje a `Barcelona` por 600€. El primer paso será encontrar la medida de similitud que utilizará el método de cómputo de similitud. En este ejemplo, el método recupera y utiliza la medida de similitud `Viajes_Measure` asociada con el concepto `Viajes`, que es el concepto más específico que subsume tanto al individuo consulta `Q1` y a `V1` (respectivamente a `V2`). La figura también representa las medidas de similitud y las tres componentes que contribuyen a ellas. En el caso de la medida `Viajes_Measure`, para la componente de similitud por posición se utiliza la instancia `idetalle` que hace referencia a la función *detalle* y para la componente de similitud por contenidos y para combinar los dos valores anteriores se utilizará la instancia `iaverage` que hace referencia

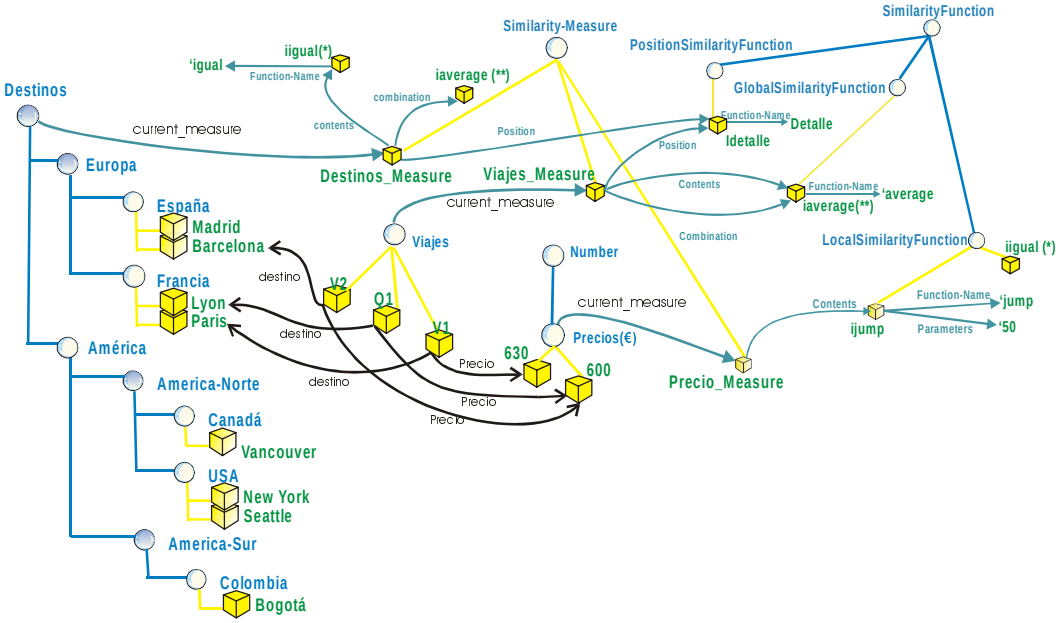


Figura 5-9. *Ejemplo de aplicación de las medidas de similitud*

a la función global que computa la *media aritmética*. Observamos el uso de una función global en la componente de contenidos, lo que significa que se tendrá en cuenta la representación estructurada del concepto `Viajes`. También se podría optar por usar una función local, lo que significaría que las instancias del concepto `Viajes` se interpretarían como valores simples y no se tendrían en cuenta sus atributos.

Como vimos en el Capítulo 4 (Apartado 5.2.1) los tipos de índice se asocian con los conceptos del dominio y definen distintas formas de medir la similitud entre individuos en función de las relaciones que contribuyen a la misma. Suponiendo un tipo de índice en el que se tienen en cuenta los dos atributos `destino` y `precio`, el método que computa la similitud aplica la medida `Viajes_Measure` para medir la similitud entre la consulta y cada uno de los dos viajes del ejemplo. La componente de contenidos (media aritmética) computa recursivamente los valores de similitud entre los rellenos de los atributos que describen el contenido de los individuos. Para comparar los rellenos del atributo destino en la consulta y el caso V1 de la Figura 5-9, `Lyon` y `París` respectivamente, utilizaremos la medida de similitud asociada al concepto más específico que contiene a ambos, es decir, la medida `Destinos_Measure` asociada al concepto `Destinos` (ya que ni el concepto `Francia` ni el concepto `Europa` tienen medidas de similitud asociadas). Observamos el uso de una función local en la componente de contenidos (`iigual`) que significa el fin de la recursión en esta rama y la comparación entre los individuos. Para tener en cuenta la posición de los individuos `Paris` y `Lyon` en la taxonomía se utiliza la componente de posición de la medida `Destinos_Measure`, que hace referencia a la función `detalle`. Ambas componentes (posición y contenidos) son combinadas usando la función `media aritmética`:

`sim(Lyon,París)=media(igual(Lyon,París),detalle(Lyon,París))=media(0,5/6)= 5/12`

Por otro lado, para comparar los rellenos del atributo `precio` se usa la medida de similitud `Precio_Measure`, cuya componente de contenidos hace referencia a la función salto (`jump`)con un parámetro de valor 50. En este caso no existe componente de posición, ni de combinación.

```
sim(600, 630) = jump(600, 630, 50) = 1
```

El valor de similitud por contenidos entre los individuos Q1 y V1 se obtiene aplicando su función de similitud por contenidos (media aritmética) a los valores de similitud obtenidos recursivamente al comparar los rellenos de sus atributos. Por otro lado, la componente de similitud por posición se computa a su vez con la función `detalle` y ambos valores se combinan a través de la `media aritmética`.

```
sim(Q1,V1)= media(media(sim(Lyon,París),sim(600,630)),detalle(Q1,V1))=
        media(media(5/12, 1), 5/6) = media(17/24, 5/6) = 0,77
```

Con el mismo procedimiento comparamos Q1 y V2:

```
sim(Q1,V2)=media(media(sim(Lyon,Barcelona),sim(600,600)),detalle(q1,v2))=
        media(media(3/8, 1), 5/6) = media(11/16, 5/6) = 73/96 = 0,76
sim(Lyon,Bcn)=media(igual(Lyon,Bcn),detalle(Lyon,Bcn))=media(0,3/4) = 3/8
sim(600,600) = jump(600, 600, 50) = 1
```

Por tanto, el método `inumeric_simcomputation_method` que resuelve la subtarea de valoración de la similitud derivada por el método de recuperación por cómputo de similitud, obtendría los valores de similitud anteriores, que son la entrada del método que resuelve la subtarea de selección de casos que se encargará de elegir el caso mejor teniendo en cuenta la información de similitud obtenidos. En este caso el método de selección elegirá en primer lugar el caso V1 para el que hemos obtenido un valor de similitud mayor.

3.1.4.5 Pautas para el diseñador

En CBROnto existen medidas de similitud predefinidas, que son genéricas e independientes del dominio y pueden ser utilizadas en cualquier aplicación. Además, para los conceptos que representan los tipos de casos el uso de una u otra puede estar recomendado según los tipos de casos predefinidos (ver Apéndice B). El diseñador de una aplicación puede utilizar alguna de las medidas predefinidas o definir medidas específicas, que se asocian a los conceptos correspondientes.

Para definir una medida de similitud se utilizarán algunas de las funciones de similitud predefinidas en CBROnto, particularizadas convenientemente. Por ejemplo, se incorporará alguna de las funciones de similitud por posición predefinidas que son genéricas pero aprovechan completamente el conocimiento del dominio subyacente con el que trabajan. Para la componente de similitud por contenidos, el diseñador también tiene la opción de usar las funciones genéricas, especialmente las funciones globales, que igual que en su uso como funciones de combinación se pueden particularizar, por ejemplo, especificando vectores de pesos. Para las funciones locales (en la componente de similitud por contenidos) se pueden usar las funciones predefinidas para tipos básicos, como números o cadenas de caracteres, aunque para los conceptos del dominio, en vez de usar funciones sencillas como la igualdad de dos individuos, es preferible particularizar las funciones predefinidas tabla o llamada a una función, especificando una tabla de valores de similitud entre cada dos valores o una función Lisp respectivamente, que sean más adecuadas para el dominio o aplicación concretos.

En este apartado hemos descrito el método de recuperación por cómputo de similitud que se basa en una aproximación computacional y la representación explícita de las medidas de similitud utilizadas. Como ya hemos comentado, los dos inconvenientes principales de este método son, por un lado, la posible degradación de la eficiencia, al computar todos los valores de similitud durante la interacción con el usuario. Este problema hace que este método no sea aplicable en bases de casos grandes. Por otro lado, y aunque se tiene en cuenta la

taxonomía de conceptos del dominio a través de la componente de posición de la medida de similitud, el método no aprovecha las oportunidades que la jerarquía ofrece desde el punto de vista de la organización de la información. El siguiente apartado describe una familia de métodos de recuperación que se basan en la clasificación de los conceptos y los individuos en la jerarquía de términos del dominio.

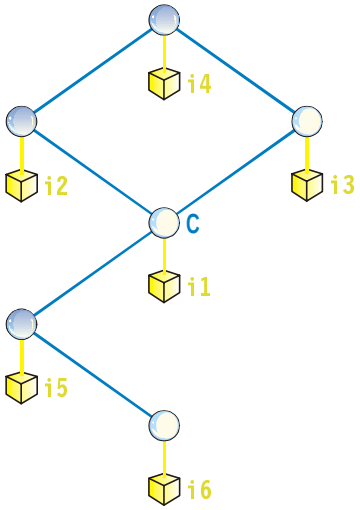
3.2 Recuperación basada en clasificación

Enmarcado dentro de este epígrafe identificamos una familia de métodos de recuperación, basados en una aproximación representacional al CBR, caracterizada porque la similitud entre la consulta y cada caso se valora según su proximidad en la estructura de organización de casos. La característica más destacable de esta aproximación es la eficiencia de los métodos de recuperación, ya que no se computan los valores de similitud en tiempo de ejecución, sino que se consulta una estructura existente. Como alternativa al cómputo exhaustivo de valores de similitud, la aproximación representacional requiere el preprocesamiento de los casos para organizarlos en una estructura adecuada que explicite una medida de similitud y que evite la búsqueda exhaustiva durante la recuperación.

Sin embargo, nuestra aproximación no se basa en precomputar estructuras optimizadas y específicas de organización de la base de casos sino en aprovechar la propia taxonomía de conceptos del dominio de la que disponemos y los mecanismos de razonamiento de las DLs, que son especialmente adecuados para este método de recuperación. Como ya hemos comentado en el *método computacional*, y así se refleja en la componente de posición de la medida de similitud, la taxonomía de conceptos influye en la similitud entre sus individuos. En los *métodos basados en clasificación* para valorar la similitud sólo intervienen los conceptos a los que pertenecen los individuos que representan a los casos. Aunque no se tienen en cuenta explícitamente los valores de sus atributos, es decir, los individuos con los que se relacionan, se tienen en cuenta implícitamente a través de los conceptos. Es decir, los valores de los atributos de un individuo influyen en su posición en la jerarquía ya que el mecanismo de reconocimiento de instancias se basa en las definiciones de los conceptos y en los asertos sobre el individuo, que incluyen sus relaciones con otros individuos.

Las variaciones relativas a los métodos que se enmarcan dentro de esta familia se refieren al uso de distintos mecanismos de la lógica descriptiva, por ejemplo, clasificación de conceptos o reconocimiento de instancias, y al uso de distintas estructuras de clasificación. Además de las ventajas debidas a la eficiencia del método, una ventaja adicional se refiere a la declaratividad que permite explicar en términos del dominio por qué los casos son similares. El inconveniente principal se debe a la dependencia de una estructura de organización que, si no es adecuada, puede no recuperar los mejores casos o no discriminar lo suficiente. Aunque los conceptos de la estructura también intervienen en el método de recuperación por cómputo de similitud, su dependencia no es tan grande porque se computa la similitud entre la consulta y todos los casos de la base de casos, al contrario que en el método basado en clasificación en el que sólo se consideran los casos del entorno de clasificación de la consulta en términos de los conceptos de la estructura.

Como se describió en el Capítulo 4, el diseñador de una aplicación puede completar el modelo del dominio original con ciertos conceptos índices que facilitan una clasificación más adecuada de los individuos para los objetivos de la aplicación. Como alternativa a la identificación manual de índices, estos conceptos también pueden identificarse utilizando técnicas inductivas como el AFC.




```
GTO (C :level -2 :direct true) = {i4}
GTO (C :level -2 :direct nil)  = {i1,i2,i3,i4,i5,i6}

GTO (C :level -1 :direct true) = {i2,i3}
GTO (C :level -1 :direct nil)  = {i1,i2,i3,i5,i6}

GTO (C :level 0 :direct true)  = {i1}
GTO (C :level 0 :direct nil)   = {i1,i5,i6}

GTO (C :level 1 :direct true)  = {i5}
GTO (C :level 1 :direct nil)   = {i5,i6}

GTO (C :level 2 :direct true)  = {i6}
GTO (C :level 2 :direct nil)   = {i6}
```

Figura 5-10. *El operador GTO*

Los siguientes apartados describen distintos métodos de recuperación por clasificación en términos de los mecanismos y la estructura que utilizan. Los Apartados 3.2.1 y 3.2.2 describen los dos métodos generales de recuperación por clasificación, el reconocimiento de instancias y la clasificación de conceptos, que se aplicarán a estructuras de organización arbitrarias. Los dos métodos se comparan en el Apartado 3.2.3. En cada aplicación concreta del método la bondad de los resultados dependerá directamente de la estructura en la que los casos están clasificados. En el Apartado 3.2.4 consideramos cómo se comportan los métodos de recuperación por clasificación cuando la estructura de organización es el retículo de conceptos formales resultante de la aplicación del AFC.

Dentro del marco genérico para valorar la similitud entre individuos (descrito en [Díaz&González01b] [Díaz&González01d]) hemos definido un operador, en términos del cuál expresaremos los métodos de recuperación por clasificación. Este operador permite una formulación genérica y configurable del tipo de estrategia que utiliza el método para recorrer las jerarquías. El operador GTO (*Generic Travel Operator*) recupera individuos de la jerarquía de subsunción. Toma como punto de partida un concepto C y el nivel al que queremos recuperar, teniendo en cuenta que los niveles positivos significan caminos descendentes en la jerarquía y los negativos significan caminos ascendentes a partir de C. Como último argumento podemos indicar si el operador debe recuperar todas las instancias de los conceptos considerados o sólo las instancias directas, es decir, que no son instancias de ninguno de sus subconceptos. Esta última es la opción por defecto. La Figura 5-10 muestra un ejemplo del operador GTO.

3.2.1 Recuperación por reconocimiento de instancias

Dentro de la aproximación representacional a la recuperación de casos usando mecanismos de las DLs, el método más intuitivo consiste en crear un individuo con las características dadas en la consulta y clasificarlo en la jerarquía de conceptos del dominio. El mecanismo de reconocimiento de instancias determina qué definiciones de conceptos son satisfechas por el individuo consulta, y recupera otros individuos clasificados de forma *similar*. Este método ha

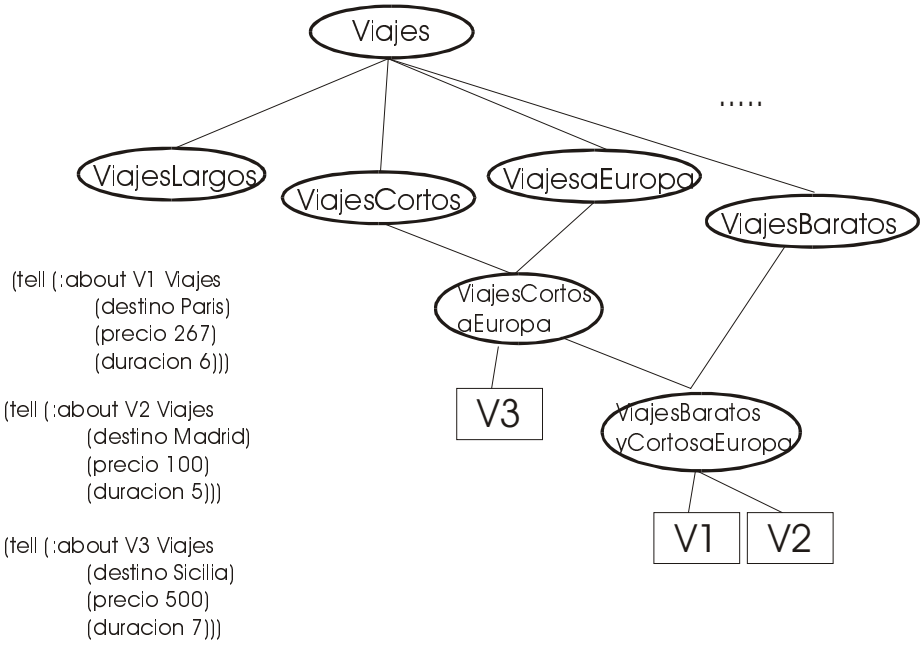


Figura 5-11. *Ejemplo de reconocimiento de instancias*

sido utilizado en otros trabajos relacionados. Por ejemplo, [Kamp96][Salotti&Ventos98] [González *et al.* 99b].

Este método de recuperación mejora la eficiencia de la búsqueda porque no se computan valores de similitud entre la consulta y los casos, sino que se aprovecha una estructura pre-computada y lleva asociado únicamente el coste del reconocimiento de instancias.

En el ejemplo representado en la Figura 5-11, la definición de los conceptos permite reconocer sus instancias en función de los valores de sus atributos. Por ejemplo, la definición de `Viajes_Cortos` indica que el sistema reconocerá como viajes cortos aquellos cuyo valor en el atributo `duración` sea menor que 10 días. Y la definición de `Viajes_Europeos` indica que el sistema reconocerá como instancias suyas aquellos individuos cuyo valor en el atributo `destino` sea una instancia del concepto `Destinos_Europeos`. Para los tres individuos dados –V1, V2 y V3– se aserta a qué conceptos primitivos pertenecen (`Viajes`) y los valores de sus atributos y el sistema infiere la posición en la taxonomía, en función de la cual se valora que V1 y V2 son más similares entre sí que cualquiera de ellos con V3.

Este método admite numerosas variaciones en función de su configuración. Por ejemplo, ¿recuperamos casos con exactamente la misma clasificación o basta con que comparta algún concepto?; si no hay ningún caso con la misma clasificación, ¿podemos generalizar para recuperar individuos que, aunque no compartan ningún concepto, estén clasificados cerca?, es decir, además de recuperar las instancias *hermanas* de la consulta ¿no podríamos recuperar también las instancias *primas*?; ¿hasta qué nivel estamos dispuestos a generalizar? Además,

dependiendo de la consulta, de la estructura conceptual y de la complejidad de los casos, ¿no podría resultar más adecuado clasificar una parte del individuo caso (por ejemplo, su descripción) en vez del caso en sí mismo? La configuración del método genérico se obtiene indicando:

- si queremos recuperar individuos clasificados exactamente igual que la consulta (semántica `and`), o basta con que comparta alguna de sus propiedades (semántica `or`).
- cómo relajar en el caso de que no haya ningún caso recuperado, es decir, cómo recorrer los niveles de la jerarquía, hacia arriba o hacia abajo y cuál es el nivel máximo (o mínimo) que consideramos como umbral de similitud aceptable.

Los siguientes pasos esquematizan el *método de recuperación por reconocimiento de instancias*:

1. Reconocer los conceptos *más específicos* {C1, .., Cn} de los que el individuo consulta es instancia.
2. En el método con semántica *and*

   $$\texttt{Resultado} = \bigcap_{i=1}^{n} \text{GTO}(\text{Ci}: \text{level}\,0: \text{direct true}) = \bigcap_{i=1}^{n} \text{instancias\_directas}(\text{Ci})$$

   En el método con semántica *or*

   $$\texttt{Resultado} = \bigcup_{i=1}^{n} \text{GTO}(\text{Ci}: \text{level}\,0: \text{direct true}) = \bigcup_{i=1}^{n} \text{instancias\_directas}(\text{Ci})$$

3. Mientras el resultado sea vacío relajar cada concepto de la forma indicada por el diseñador a través de los parámetros `level` y `direct` del operador GTO y obtener el conjunto de instancias resultado.
4. De entre estas instancias, seleccionar el conjunto de casos que se recuperan.

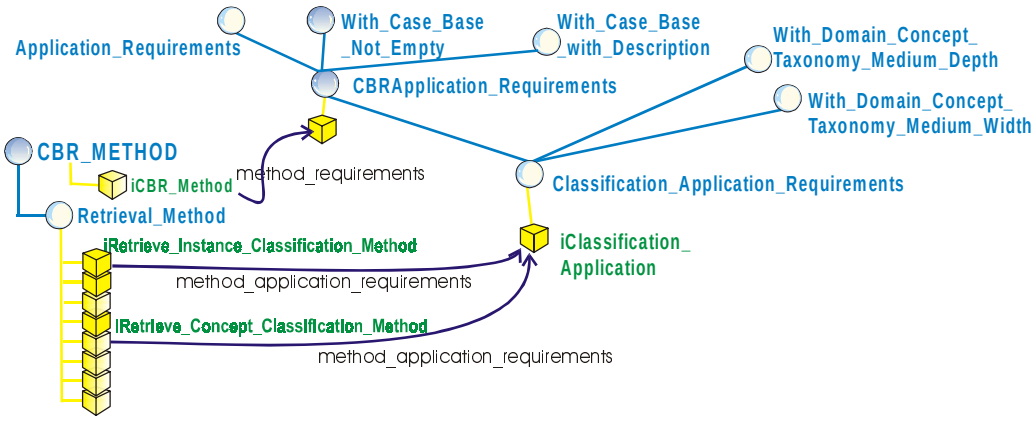
Aunque se podría formalizar con un único método configurable, hemos decidido incluir dos PSMs en la biblioteca de métodos, uno para la semántica *and* y otro para la semántica *or*. Observamos que el resultado de cada uno de los métodos se obtiene invocando una cierta secuencia de llamadas al operador GTO cuyos resultados se combinan por unión o por intersección. La secuencia de llamadas concreta que se llevará a cabo la configurará el diseñador del sistema a través de los requisitos –de diseño o de conocimiento– del método. Después de la descripción de los requisitos del método en el Apartado 3.2.1.1, el Apartado 3.2.1.2 indica cómo resolver las subtareas derivadas de la aplicación de este método.

3.2.1.1 Requisitos del método

Los métodos de recuperación basados en clasificación tienen en común ciertas características, en concreto respecto a sus requisitos, por lo que aprovechamos la jerarquía de subsunción y la herencia para representarlos. Los métodos de recuperación por clasificación se basan en la existencia de una taxonomía de conceptos del dominio suficientemente poblada, como indica el concepto `Classification_Application_Requirements`, que se muestra en la Figura 5-12, y que representa los requisitos de aplicabilidad comunes a todos los métodos basados en clasificación.

Los requisitos de entrada se dividen en tres componentes:

- Como requisito paramétrico el método recibe, igual que el resto de los métodos de recuperación, la descripción de la consulta (atributo `query`).
- Como requisitos de diseño –`iRecognition_Design`– el método recibe:




```
(defconcept Classification_Application_Requirements
:is-primitive (:and CBRApplication_Requirements
  (With_Domain_Concept_Taxonomy_Medium_Depth)(With_Domain_Concept_Taxonomy_Medium_Width)))

(tell (:about iRecognition_Application Classification_Application_Requirements))
(tell (:about iClassification_Application Classification_Application_Requirements))
```

Figura 5-12. *Requisitos de aplicabilidad*

– en el atributo `relation_path` una cadena de relaciones para indicar que lo que queremos clasificar es alguna de las componentes del individuo consulta. En el caso particular de que la cadena de relaciones sea vacía, el individuo que clasificaremos es el propio individuo consulta. La cadena de relaciones es de la forma `((r1 l1) (r2 l2) (r3 l3) .....(rm lm))`, donde ri son relaciones y li son números (positivos o negativos) que pueden incluirse o no para representar el máximo (o mínimo) nivel hasta el que generalizaremos las relaciones del camino. Este mecanismo se utiliza cuando la taxonomía en base a la cual se hace la recuperación no es directamente la zona en la que se clasifica la consulta sino otra zona de la jerarquía que es accesible a través de sus atributos.

– en el atributo `gto_specification` una lista de pares de valores que indican la secuencia de llamadas al operador GTO que lleva a cabo el método. Por ejemplo, la entrada `((0,true) (1,true) (2,true) (0,false))` en el método de semántica *and* generaría la secuencia de llamadas siguiente:

$$\texttt{Llamada inicial} \rightarrow \texttt{Resultado} = \bigcap_{i=1}^{n} \mathrm{GTO}(\mathrm{Ci}:\mathrm{level}\,0:\mathrm{direct\ true})$$

Mientras el resultado sea vacío relajar la búsqueda, es decir, hacemos la llamada asociada con la siguiente entrada de la lista dada en el atributo `gto_specification`:

$$\texttt{Primera vuelta} \rightarrow \texttt{Resultado} = \bigcap_{i=1}^{n} \mathrm{GTO}(\mathrm{Ci}:\mathrm{level}\,1:\mathrm{direct\ true})$$

$$\texttt{Segunda vuelta} \rightarrow \texttt{Resultado} = \bigcap_{i=1}^{n} \mathrm{GTO}(\mathrm{Ci}:\mathrm{level}\,2:\mathrm{direct\ true})$$

$$\texttt{Tercera vuelta} \rightarrow \texttt{Resultado} = \bigcap_{i=1}^{n} \mathrm{GTO}(\mathrm{Ci}:\mathrm{level}\,0:\mathrm{direct\ false})$$

- Como requisitos de conocimiento, el método permite un mecanismo adicional de configuración de la secuencia de llamadas al operador GTO. Este mecanismo con-

siste en realizar anotaciones en los conceptos del dominio que especifican la forma de relajar un concepto que interviene en la recuperación por clasificación. El individuo que se anota en el concepto es una instancia del concepto `GTO_Specification` de CBROnto e indica (en el atributo `level`) el nivel máximo (número positivo o negativo) que permito alcanzar, si permito también recuperar instancias no directas (en el atributo `direct`) y el orden de generalización (en el atributo `order`). Por ejemplo, una forma alternativa para obtener los resultados de la secuencia explícita del ejemplo, la instancia indicará: `level +2, order level_first, direct relax`. Esto significa que comenzaremos a relajar el concepto inicial (nivel 0) hasta el nivel 2 y después relajaremos la condición de instancias directas.

Esta segunda forma de especificar cómo relajar la búsqueda esta justificada cuando el modo de relajar depende de la naturaleza del concepto (por ejemplo, conceptos que representan precondiciones u objetivos –ver Apartado 3.2.4.3).

3.2.1.2 Resolución de las subtareas

El método de recuperación por reconocimiento de instancias resuelve la tarea de recuperación descomponiéndola en dos subtareas: `AssessSim_Task` (instancia `irecognize_task`) y `Select_Task` (instancia `iselect_case`).

La subtarea `irecognize_task` tiene como objetivo la recuperación de las instancias similares a una dada según los conceptos bajo los que se haya clasificado. Existen dos métodos básicos cuya competencia es adecuada para resolver la tarea `irecognize_task`, y ambos reciben como parte de sus requisitos paramétricos el individuo consulta (`query`) y como parte de sus requisitos de diseño una cadena de relaciones (atributo `relation_path`):

1. Usando la cadena de relaciones (r1, ..., rm) a partir del individuo `query` obtenemos un individuo[6] q_i que clasificamos en la jerarquía de conceptos.
2. El individuo q_i se reconoce como instancia de los conceptos C1, ..., Cp.
   - El método con semántica and (`iinstance_classification_and`) recupera los individuos $caso_i$ que están clasificados bajo *todos* los conceptos C1, ..., Cp.
   - El método con semántica or, `iinstance_classification_or` recupera los individuos $caso_i$ que están clasificados bajo *alguno* de los conceptos C1, ..., Cp.
3. En ambos casos se recuperan como casos similares a la consulta `query` los individuos (casos) que se alcanzan al seguir el camino de relaciones inversas partiendo de los individuos $caso_i$.

Ninguno de los dos métodos añade requisitos adicionales (aparte de los heredados).

Si el conjunto de resultados es vacío se procede a la generalización (Figura 5-13). El diseñador puede configurar la generalización usando el atributo `generalize_first` del individuo que representa los requisitos de diseño de los métodos `iinstance_classification_and` e `iinstance_classification_or`. Los valores posibles de este atributo son:

- El valor `relation` indica que el método relaja primero las relaciones del camino –especificado en el atributo `relation_path` de los requisitos de diseño– nivel por nivel de todas las relaciones `ri` en las que se especifique un valor `li`, manteniendo el nivel en los conceptos.

[6] Realmente si alguna de las relaciones del camino es multivaluada obtendremos un conjunto de individuos. Suponemos que todas son univaluadas para simplificar la exposición.

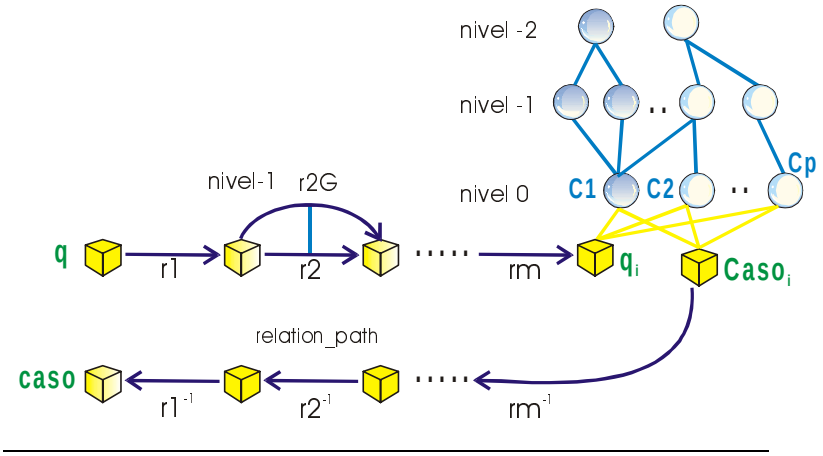


Figura 5-13. *Acceso a una zona de clasificación*

- El valor `concept` (comportamiento por defecto del método) indica que el método generaliza primero los conceptos hasta el nivel máximo –especificado en el atributo `gto_specification` de los requisitos del diseño del método o como anotaciones de los conceptos– y posteriormente de las relaciones del camino usando los niveles (`li`) que se especifican para cada relación `ri`.
- El valor `alternate_relation_first`, indica que el método generaliza un nivel de las relaciones y otro de los conceptos, hasta que el resultado sea no vacío o alcance los niveles máximos de generalización.
- El valor `alternate_concept_first`, es equivalente pero generaliza los conceptos en primer lugar.

Después de resolver la tarea `irecognize_task` se resuelve la tarea `iselect_case`, cuyo objetivo es el de seleccionar el caso mejor teniendo en cuenta la información de similitud obtenida por el método que resuelva la tarea anterior.

Se puede observar que esta tarea se representa mediante un individuo distinto al utilizado para representar la subtarea de selección del método de recuperación por cómputo de similitud `iselect_best`. Esto se debe a que, además de los métodos de selección que resuelven la tarea `iselect_best` –`iuserselectcase_method`, `iselectallcases_method` e `iselectmaxcase_method`– la tarea `iselect_case` puede ser resuelta por dos métodos adicionales:

1. `iselect_computation_method` computa la similitud entre la consulta y cada uno de los casos que haya sido recuperado por clasificación en la tarea anterior, y selecciona uno de ellos (resolución de la tarea `iselect_best`). Este método se corresponde con el método de recuperación por cómputo de similitud (pero no resuelve la primera subtarea `iobtain_cases`) y resuelve la tarea `iselect_case` pero no `iselect_best` para evitar la posibilidad de generar una configuración de tareas y métodos con recursión infinita, al elegir el mismo método como método de recuperación y de selección.
2. `iretrieve_relcriteria_method` selecciona los casos que satisfagan un criterio de relevancia dada. Este método se corresponde con el método de recuperación de casos aplicando criterios de relevancia que se describe en el Apartado 3.3.

Este uso de distintos individuos para representar a las tareas permite controlar el principal inconveniente del método computacional, y reducir su coste al emplear un método de bús-

queda exhaustiva sobre una base de casos reducida que sólo incluye los casos clasificados de forma *similar* a la consulta.

3.2.2 Recuperación por clasificación de conceptos

Dentro de la aproximación representacional a la recuperación de casos utilizando mecanismos de las DLs, un método alternativo al reconocimiento de instancias consiste en crear un *concepto* con las características dadas en la consulta y clasificarlo en la jerarquía conceptual para recuperar los individuos que se reconozcan como instancias suyas. Este método se ha utilizado, por ejemplo, en [Kamp96] y [Coupey *et al.* 98]. Los siguientes pasos esquematizan el *método de recuperación por clasificación de conceptos*:

Definir un concepto Cq con las características dadas en la consulta (atributo `query` de los requisitos paramétricos) y clasificar el concepto que se encuentra al final de la cadena de relaciones (que recibe en el atributo `relation_path` de los requisitos de diseño), comenzando por dicho concepto.

1. Obtener el conjunto de conceptos {C1, .., Cn} índices más específicos de los que el concepto clasificado es subconcepto.
2. Recuperar la unión de las instancias de los conceptos {C1,...,Cn}. Como cada uno de los conceptos representa una abstracción de un cierto número de propiedades que las instancias tienen en común, recuperaremos la unión de todas ellas:

   $$\texttt{Resultado} = \bigcup_{i=1}^{n} \mathrm{GTO}(\mathrm{Ci}: \mathrm{level}\,0: \mathrm{direct\ true}) = \bigcup_{i=1}^{n} \mathrm{instancias\_directas}(\mathrm{Ci})$$

3. Si el resultado es vacío, igual que en el método de recuperación de casos por reconocimiento de instancias, se utilizarán las especificaciones del diseñador para *refinar* los conceptos {C1,...,Cn} y encontrar casos parcialmente similares a la consulta.
4. Recorrer la cadena de relaciones inversas para acceder a los casos recuperados –que serán las instancias de los conceptos que obtenemos al final del proceso.
5. Seleccionar de entre los candidatos cuáles serán los casos recuperados. Todos los candidatos tendrán en común ciertas propiedades representadas por los conceptos comunes de los que son instancia. Serán más similares los casos que compartan un mayor número de conceptos. Además la subtarea de selección puede tener en cuenta otras propiedades discriminantes.

3.2.2.1 Resolución de las Subtareas

El método de recuperación por clasificación de conceptos –que se representa con el individuo canónico `iRetrieve_concept_classification_method`– comparte los requisitos con el método de recuperación de casos por reconocimiento de instancias y resuelve la tarea de recuperación de casos descomponiéndola en dos subtareas: valoración de la similitud –`AssessSim_Task` (instancia `iclassify_task`)– y selección de casos –`Select_Task` (instancia `iselect_case`).

La subtarea `iclassify_task` tiene como objetivo la definición de un concepto que represente las características de la consulta que recibe el método de recuperación en el atributo `query` de los requisitos paramétricos. Además, la subtarea se encarga también de clasificar el concepto que representa la consulta en la taxonomía de conceptos del dominio.

La tarea representada por el individuo canónico `iclassify_task` está ligada a `iclassify_method`, el único método de CBROnto cuya competencia es adecuada para resolverla. En

general el método `iclassify_method` recupera como individuos similares a la consulta aquellos individuos que son instancia del concepto que representa a la consulta –siguiendo la cadena de relaciones para elegir la zona de la taxonomía en la que usamos la clasificación. El método `iclassify_method` es un método de resolución, que no genera subtareas. Después de la tarea `iclassify_task` se resuelve la tarea `iselect_case` (que ha sido descrita en el método de recuperación por reconocimiento de instancias Apartado 3.2.1.2).

En el siguiente apartado comparamos los dos métodos descritos dentro de este epígrafe. De nuevo resaltamos el hecho de que la bondad de los resultados de aplicación de ambos métodos de recuperación depende directamente de la estructura de organización. El Apartado 3.2.4 describe el comportamiento del método de recuperación por reconocimiento de instancias cuando la estructura de organización es el retículo de conceptos formales obtenido por la aplicación del AFC.

3.2.3 Comparación entre las dos aproximaciones

Como describimos en el Capítulo 2, en general, e independientemente del grado de expresividad que tenga una DL, la expresividad de su lenguaje asertivo suele ser más reducida que la de su lenguaje terminológico. Esto es cierto también para el sistema LOOM y, por tanto, la definición de consultas como conceptos ofrece una expresividad mayor que la definición de consultas como instancias [MacGregor91]. Sin embargo, existen varias razones que dan soporte a la definición de consultas como individuos y no como conceptos. En primer lugar la suposición del diseño general de los sistemas terminológicos de representación, según la cual la componente terminológica de una base de conocimiento es básicamente estática, mientras que la componente asertiva es dinámica y susceptible de cambios (asertos y retracciones) que varían la posición del individuo en la jerarquía. Los cambios en los individuos no afectan para nada la jerarquía de clasificación de conceptos, ya que la semántica de los sistemas de DLs, y en particular la del sistema LOOM, se diseñó para que la jerarquía de conceptos sea inmune a cambios en los individuos. La jerarquía cambiaría de forma monótona si añado una nueva definición de concepto, pero nunca cambiará la posición de un concepto existente. Esto tiene una repercusión directa en la forma de razonar del sistema. En concreto, si una operación de clasificación de un concepto C_2 bajo otro C_1 depende de un cambio en un individuo i, no se lleva a cabo previendo que el individuo i pueda cambiar. Los mecanismos de razonamiento sólo hacen las operaciones de clasificación "estables" que no dependan de cambios en individuos. Por esta razón si la consulta es un concepto según como esté construida puede que alguna de las clasificaciones (aunque sea posible) no se haga correctamente y no recuperemos ciertos casos que serían adecuados.

Además, el método que define las consultas como individuos permite utilizar mecanismos que completan las consultas que no son adecuados cuando la consulta se representa como un concepto. El mecanismo utiliza reglas asociadas a los conceptos. Aunque en teoría, este mecanismo permitiría asociar una regla con el concepto que representa a la consulta, en la práctica el mecanismo no funciona porque las reglas no se "disparan" hasta que se encuentra un individuo que sea instancia de ese concepto.

Por ejemplo, supongamos que el modelo del dominio se ha definido la regla (`implies` C_1 C_2) que expresa que cualquier individuo que se reconozca como instancia de C_1, también se reconoce como instancia de C_2. Si definimos una consulta Q como un concepto, el sistema puede clasificar correctamente Q como un subconcepto de C_1 ($Q \subseteq C_1$) pero no puede inferir ninguna relación entre Q y C_2.

Sin embargo, si Q se define como un individuo, el sistema puede reconocer que Q es instancia del concepto C_1 y también que es instancia del concepto C_2. En este sentido, la consulta dada por el usuario ha sido completada con conocimiento adicional que forma parte del modelo del dominio.

3.2.4 Recuperación en el retículo de conceptos formales

En este apartado describimos como se comporta el método de recuperación de casos basado en el reconocimiento de instancias cuando la estructura de organización de los casos es el retículo resultante de la aplicación del Análisis Formal de Conceptos (descrito en el Capítulo 4 (Apartado 5.3)). Vamos a distinguir dos métodos que dependen de cómo se construya el retículo, es decir, cuáles son las propiedades que utilicemos como atributos del contexto formal. En el primer método se utilizan como atributos del contexto formal ciertas propiedades que describen a los casos y que se especifican durante la fase de diseño del sistema. El segundo método sólo es aplicable cuando los casos se describen en términos de las precondiciones necesarias y/o los objetivos que satisface su solución.

Para cualquiera de los métodos se requiere la aplicación previa del AFC para construir los retículos de conceptos formales que estructuran la base de casos.

3.2.4.1 Métodos de construcción de los retículos de conceptos formales

Como se describió en el Capítulo 4 (Apartado 3.1.2) CBROnto define varias tareas de primer nivel –entre las que se encuentra la tarea de resolución de problemas `CBR_Task`. La resolución de estas tareas debe invocarse de forma externa. Entre estas tareas se encuentran también la adquisición de casos y de conocimiento del dominio, la integración del conocimiento del dominio con el conocimiento de CBROnto y la organización de la base de casos en una estructura que facilite la recuperación de casos.

En la versión actual de CBROnto, existen dos métodos cuya competencia es adecuada para resolver la tarea de organización de la base de casos (`CB_Organization_Task`): `iFCA_GoalPre_Method`, `iFCA_Properties_Method`. Ambos métodos se basan en la aplicación del Análisis Formal de Conceptos a los casos, aunque difieren en los atributos utilizados para definir el contexto formal.

El método `iFCA_Properties_Method` construye un retículo de organización de los casos y extrae las reglas de dependencia basándose en ciertas propiedades descriptivas de los casos (como se describió en los Apartados 5.3.3.1 y 5.3.3.4 del Capítulo 4). El método recibe como parte de sus requisitos de diseño un atributo `relation_path` que especifica las cadenas de relaciones que se utilizarán para definir el contexto formal. Recordamos que en el ejemplo de construcción del retículo sobre casos estructurados descrito en el Apartado 5.3.3.4 del Capítulo 4, se añadían al conjunto M de atributos del contexto formal todos los caminos parciales de relaciones que llevan desde el individuo caso hasta individuos (que pueden ser terminales o no). Cada descriptor en *M* incluye un camino (posiblemente incompleto) de relaciones y el individuo (interno o terminal) que se alcanza desde el individuo caso al seguir el camino de relaciones. La especificación de las cadenas de relaciones por parte del diseñador es opcional pero si se incluyen permiten que el diseñador filtre cuáles de todos estos caminos parciales se usarán realmente en la construcción del retículo.

El método `iFCA_GoalPre_Method` construye los retículos basándose en los objetivos y precondiciones de los casos según se describió en el Apartado 5.3.3.3 del Capítulo 4. Este método incluye como requisito de aplicabilidad que los casos estén descritos según sus precondiciones y objetivos (la base de casos especificada en el contexto (`case-base`) debe ser un

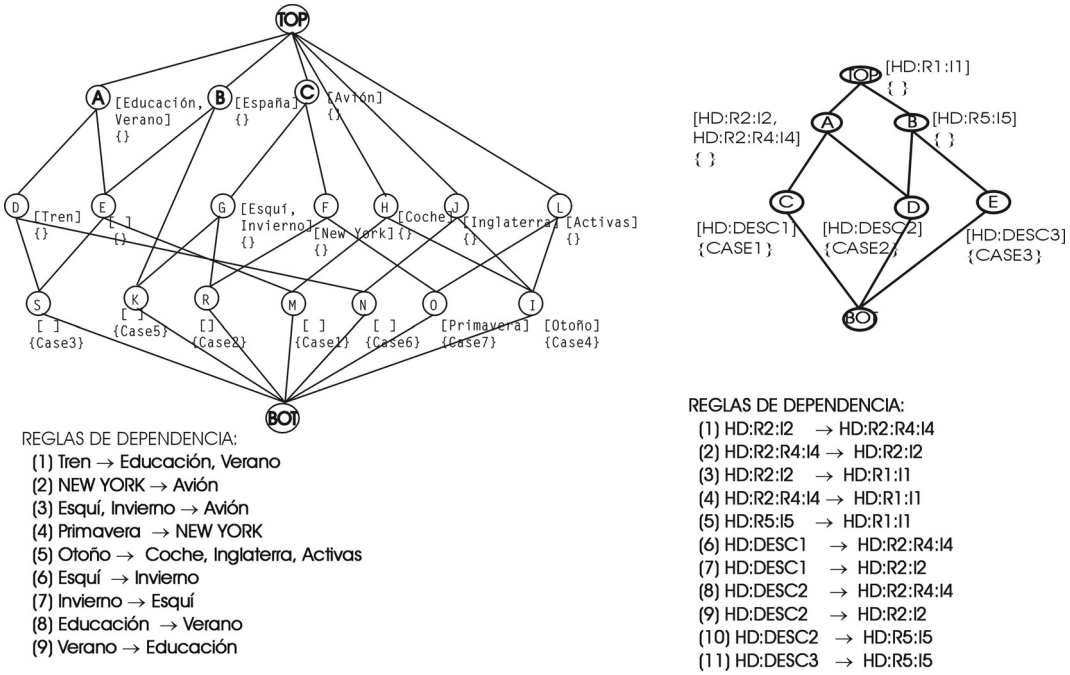


Figura 5-14. *Retículos de conceptos formales y reglas de dependencia*

individuo instancia de `Case_With_Goals` y `Case_With_Pre`). Como requisito de conocimiento (y también de aplicabilidad) se pide que los conceptos que definen los objetivos en el dominio y las propiedades que se usan como precondiciones se hayan clasificado bajo los conceptos `Goal` y `Precondition` de CBROnto, que son con los que trabaja el método.

3.2.4.2 Método de recuperación de casos por compleción de consultas

En este apartado describimos el método de recuperación de casos por reconocimiento de instancias en el retículo de conceptos formales –representado por el individuo canónico `iCase_Inspection_Method` [Díaz&González00b] [Díaz&González01b]. Este método es un caso particular del método de recuperación por reconocimiento de instancias que resulta adecuado para recuperar casos que están organizados en la estructura de retículo de conceptos formales construida previamente por el método `iFCA_Properties_Method.` Como ejemplo, trabajaremos con los retículos y los conjuntos de dependencias construidos en los Apartados 5.3.3.1 y 5.3.3.2 del Capítulo 4 que se muestran en la Figura 5-14.

Las reglas de dependencia detectan regularidades de coaparición de atributos que satisfacen todos los casos de la base. Este método se basa en estas reglas para completar las consultas del usuario y definir un método de recuperación exacta, en el sentido de que los casos recuperados satisfacen exactamente los requisitos de la consulta completada. La propia base de casos guía el proceso de formulación de "buenas" consultas, que son aquellas para las que hay casos que las cumplen exactamente.

Este método de recuperación de casos es un método interactivo donde el usuario proporciona ciertos descriptores para la consulta, mientras que el sistema propone otros que deduce de los proporcionados por el usuario. Para la deducción utiliza las reglas de dependencia extraídas de la base de casos durante el proceso de aplicación del AFC. El objetivo de este

proceso es guiar la consulta hacia los conceptos más específicos del retículo y recuperar los casos que satisfacen todos los descriptores de la consulta.

La resolución del método aplica de forma cíclica los dos pasos siguientes:

- Un proceso interactivo de construcción de la consulta usando las reglas de dependencia que se basa en:
  - Construir un individuo que representa la consulta y clasificarlo aprovechando el mecanismo de reconocimiento automático de instancias.
  - Cada regla de dependencia se formaliza como un concepto cuya condición suficiente es la parte derecha de la regla, y cuya condición necesaria (utilizada para reconocer al individuo consulta) es la parte izquierda de la regla.
  - De esta forma, es el propio proceso de reconocimiento de instancias el que determina qué reglas son aplicables, ya que el individuo consulta se clasifica en los conceptos adecuados.
- Una vez que la consulta se da por terminada se accede a los conceptos más específicos del retículo de conceptos formales, bajos los que se clasifica la consulta para recuperar los individuos clasificados bajo los mismos conceptos que el individuo consulta.

Resolución de las subtareas

El método `iCase_Inspection_Method` resuelve la tarea de recuperación descomponiéndola en dos subtareas: `AssessSim_Task` (a través de la instancia `iinspectiontask`) y `Select_Task` (a través de la instancia `iselect_case` descrita en el Apartado 3.2.1.2).

La subtarea `iinspectiontask` está ligada a `iinspection_method`, un método que se encarga de completar la consulta usando las reglas de dependencia, y clasificarla en el retículo para recuperar casos. El método descompone la tarea `iinspectiontask` en las dos subtareas siguientes:

- `iquery_completiontask` asociada al método de resolución interactivo `iquery_completionmethod` que se encarga de construir y completar la consulta del usuario usando las reglas de dependencia extraídas de la aplicación del AFC a los casos.
- `iquery_FCAretrieval_task` asociada al método de resolución `iquery_FCAretrieval_method` que se encarga de recuperar los individuos clasificados bajo los conceptos más específicos del retículo AFC bajo los que se clasifica el individuo consulta.

Requisitos del método

El método incluye como requisito de aplicabilidad adicional la construcción previa de la estructura de retículo de conceptos formales por el método `iFCA_Properties_Method` lo que supone también la definición de las reglas de dependencias. El atributo `case-organization-structure` del individuo que describe el contexto de aplicación actual tendrá el valor `FCA_lattice`, cuando el método `iFCA_Properties_Method` haya construido el retículo.

El método no incluye requisitos de conocimiento adicionales. Los requisitos paramétricos no tienen sentido porque este método se aplica durante la fase de diseño del sistema.

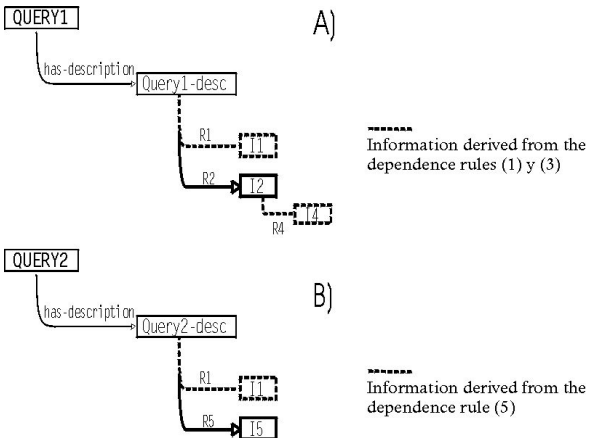


Figura 5-15. *Ejemplo de consultas para recuperación*

Ejemplos de aplicación del método

- ***Ejemplo 1***

En el ejemplo de la agencia de viajes, al que corresponde el retículo de la Figura 5-14 (izquierda), el usuario construye una consulta seleccionando ciertos descriptores, por ejemplo, `Tipo_de_Vacaciones:Esquí`. El método detecta una dependencia (regla 6) aplicable entre el descriptor `Tipo_de_Vacaciones:Esquí` y `Estación:Invierno`, que significa que en esta base de casos en particular, no hay ningún viaje a esquiar que no sea en invierno. Por tanto, se completa la consulta para guiarla hacia esos casos. Las reglas aplicadas son:

(6) `Tipo_de_Vacaciones:Esquí` → `Estación:Invierno`
(3) `Tipo_de_Vacaciones:Esquí, Estación:Invierno` → `Transporte:Avión`

y la consulta completada es la siguiente:

`Tipo_de_Vacaciones: Esquí`
`Estación: Invierno`
`Transporte: Avión`

En este punto el usuario puede elegir entre terminar el proceso interactivo de formulación de la consulta o dar nuevos descriptores. Tras este proceso se crea un individuo con las características dadas que se clasificará en el retículo de la Figura 5-14 (izquierda) bajo el concepto G. Los casos que resultan del proceso de recuperación son los casos de la extensión del concepto formal G, es decir, `Caso2` y `Caso5`.

El usuario puede continuar el proceso, eligiendo nuevos descriptores para la consulta. El sistema guiará el proceso hacia los conceptos más específicos, es decir, K o R, por lo que el conjunto de descriptores ofrecidos al usuario se reduce a: `[Destino:Nueva York; Destino:España]`.

- ***Ejemplo 2***

En el dominio genérico utilizado para construir el retículo de la Figura 5-14 (derecha), supongamos que el usuario plantea las consultas representadas en la Figura 5-15. Las líneas continuas representan los descriptores dados por el usuario, mientras que las líneas disconti-

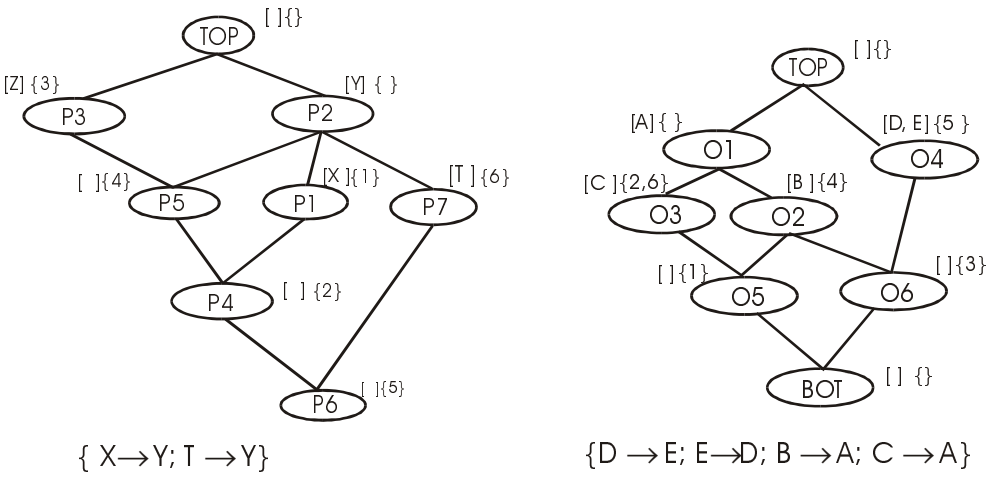


Figura 5-16. *Retículos de precondiciones y objetivos*

nuas representan la información inferida por el sistema utilizando las reglas de dependencias de la Figura 5-14 (derecha).

En la consulta de la Figura 5-15 (A) el usuario establece que el individuo que representa la descripción del caso tiene como valor de la propiedad (atributo o relación) `R2` el relleno `I2`, y a la consulta se le asigna el descriptor `[HD:R2:I2]`. El individuo consulta se completa usando las reglas de dependencia (1) y (3) de la Figura 5-14 (derecha) y clasificado bajo el concepto A del retículo, al satisfacer las propiedades de su intensión `[HD:R2:I2, HD:R2:R4:I4, HD:R1:I1]`; aunque sólo la primera de ellas ha sido dada por el usuario.

El resultado de la recuperación son los casos de la extensión del concepto A, es decir, `Caso1` y `Caso2`. La aplicación posterior de un método de selección de casos que realice una valoración adicional de similitud, ordenaría los casos para seleccionar `Caso1` como el caso más similar porque tiene menos descriptores no pedidos que el caso `Caso2`.

En la consulta de la Figura 5-15 (B) el usuario establece que la consulta tiene I5 como valor de la propiedad R5, la consulta es completada usando la regla de dependencia (5) y clasificada bajo el concepto B del retículo, porque satisface las propiedades de su intensión `[HD:R5:I5, HD:R1:I1]`. El método recupera los casos de la extensión del concepto B es decir, `Caso2` y `Caso3`. El proceso de selección elegirá `Caso3` como el más similar.

3.2.4.3 Método de recuperación por precondiciones y objetivos

En este apartado describimos el método de recuperación de casos por reconocimiento de instancias en los retículos de precondiciones y objetivos –representado por el individuo canónico `iRetrieve_GoalPre_Method` [Díaz&González01c]. Es un caso particular del método de recuperación por reconocimiento de instancias que resulta adecuado para recuperar casos que estén descritos por los objetivos que satisface su solución y/o por las propiedades que definen la precondición que debe satisfacer la situación en la que aplicamos la solución descrita en el caso. Este tipo de casos los encontramos típicamente en sistemas de planificación en los que, dado un conjunto de objetivos, el planificador busca un plan que sea aplicable a la situación descrita en la consulta y que obtenga (en la medida de lo posible) los objetivos pedidos [Hammond89].

El método se basa en los retículos de conceptos formales de precondiciones y objetivos que el método `iFCA_GoalPre_Method` habrá construido previamente. Supongamos que disponemos de los retículos construidos en el Capítulo 4 (Apartado 5.3.3.3), que se muestran en la Figura 5-16. Este método de recuperación de casos proporciona varias posibilidades de interacción con el usuario, que puede proporcionar un conjunto de objetivos para encontrar casos que los consigan, un conjunto de propiedades, para encontrar casos aplicables, o ambas cosas, para encontrar casos aplicables que satisfagan los objetivos pedidos.

Sean cuales sean las características dadas, el sistema las utilizará para crear un individuo consulta con dos componentes: un individuo (`query_goal`) sobre el que asertamos los objetivos de la consulta, y un individuo (`query_desc`) sobre el que asertamos las propiedades que satisface la situación actual. El sistema de DLs clasificará automáticamente los individuos en los retículos.

Resolución de las subtareas

Al igual que el método de recuperación por reconocimiento de instancias, el método `iRetrieve_GoalPre_Method` resuelve la tarea de recuperación descomponiéndola en dos subtareas: valoración de similitud –`AssessSim_Task` (pero a través de la instancia `irecognizeGoalPre_task`)– y selección –`Select_Task` (a través de la misma instancia `iselect_case` ya descrita).

La subtarea `irecognizeGoalPre_task` tiene como objetivo la recuperación de las instancias similares a una dada según los conceptos de los retículos de objetivos y precondiciones bajo los que se haya clasificado. Existen tres métodos cuya competencia es adecuada para resolver la tarea `irecognizeGoalPre_task` que se describen en los siguientes apartados (también en [Díaz&González01c]):

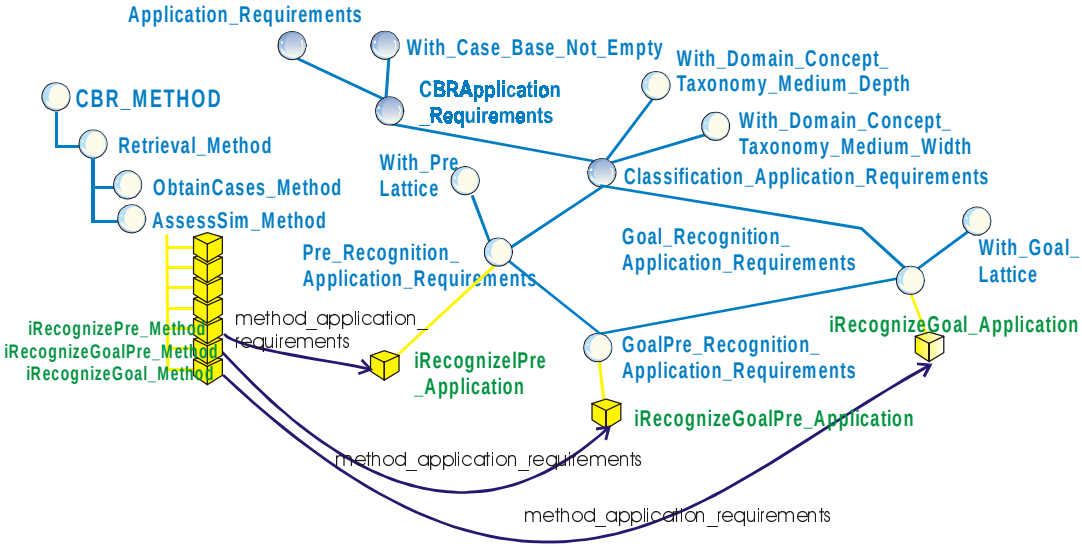
- El método `irecognizeGoal_Method` se basa en clasificar un individuo que representa los objetivos pedidos por la consulta en el retículo de objetivos.
- El método `irecognizePre_Method` se basa en clasificar un individuo que representa la situación actual descrita por la consulta en el retículo de precondiciones.
- El método `irecognizeGoalPre_Method` se basa en clasificar el individuo de objetivos de la consulta en el retículo de objetivos, y el individuo de propiedades en el retículo de precondiciones y combinar los resultados de clasificación obtenidos.

Requisitos de los métodos

Como el método `irecognizeGoal_Method` basa la recuperación en el retículo de objetivos incluye como requisito de aplicabilidad haber definido previamente el retículo de objetivos, es decir, la aplicación del AFC utilizando como atributos del contexto formal los objetivos que satisfacen los casos. En la Figura 5-17 se muestra la estructura de los requisitos de aplicabilidad del método `irecognizeGoal_Method`. De forma análoga el método `irecognizePre_Method` tendrá como requisito de aplicabilidad la definición del retículo de precondiciones, y el método `irecognizeGoalPre_Method` tendrá como requisito de aplicabilidad la definición previa de ambos retículos: el de precondiciones y el de objetivos.

Los atributos `goal-organization-structure` y `pre-organization-structure` que aparecen en las definiciones de los conceptos de la Figura 5-17 son parte de los que describen el contexto actual que se utiliza para determinar la aplicabilidad de los métodos (se describe en el Capítulo 6), y su relleno (el individuo `FCA_lattice`) es asertado cuando el método `iFCA_GoalPre_Method` construye los retículos de objetivos y precondiciones respectivamente.

Respecto a los requisitos de entrada, los métodos reciben como parte de sus requisitos paramétricos (heredados del método CBR) el individuo consulta (atributo `query`). En estos




```
(defconcept Pre_Recognition_Application_Requirements
    :is-primitive (:and Classification_Application_Requirements (With_Pre_Lattice)))
(defconcept Goal_Recognition_Application_Requirements
    :is-primitive (:and Classification_Application_Requirements (With_Goal_Lattice)))
(defconcept GoalPre_Recognition_Application_Requirements
    :is-primitive (:and Classification_Application_Requirements (With_Pre_Lattice)(With_Goal_Lattice)))
(defconcept With_Goal_Lattice :is  (:and (:the Goal-organization-structure lattice)))
(defconcept With_Pre_Lattice :is  (:and (:the Pre-organization-structure lattice)))
(defconcept With_GoalPre_Lattice :is  (:and With_Goal_Lattice With_Pre_Lattice))
```

Figura 5-17. *Requisitos de Aplicabilidad de los métodos de recuperación basados en precondiciones y objetivos*

métodos los requisitos paramétricos se especializan para indicar que la consulta estará descrita en términos de sus precondiciones y objetivos, es decir, la consulta es una instancia de los conceptos `Query_with_Goals` y/o `Query_with_Pre`, lo que garantiza el correcto funcionamiento del método. Ninguno de los métodos añade nuevos requisitos de diseño, ni de conocimiento.

```
(defconcept Goal_Recognition_Parameter :is-primitive (:and CBR_Parameter
                                        (:the query Query_with_Goals)))
(defconcept Pre_Recognition_Parameter :is-primitive (:and CBR_Parameter
                                        (:the query Query_with_Pre)))
(defconcept GoalPre_Recognition_Parameter :is-primitive
            (:and CBR_Parameter
            (:the query Query_with_Goals) (:the query Query_with_Pre)))
```

Recuperación sobre el retículo de objetivos

El método de recuperación sobre el retículo de objetivos –representado por el individuo canónico `irecognizeGoal_Method`– basa la recuperación en el retículo de objetivos e intentará encontrar casos que satisfagan todos los objetivos pedidos (y ninguno más). El método se representa como un método de resolución que obtiene el conjunto de conceptos más específicos del retículo de objetivos {O1,..,On} bajo los que se clasifica el individuo `query_goals`;

y recupera las instancias directas de todos los Oi. Este proceso se corresponde con un tipo de recuperación exacta. Si no se obtienen resultados, el paso de recuperación aproximada encuentra, en primer lugar, los casos que satisfacen todos los objetivos pedidos aunque otros (no pedidos) también se cumplan. Si el paso anterior tampoco obtiene resultados se buscará, en segundo lugar, los casos que maximicen el número de objetivos pedidos aunque alguno de ellos no se cumpla. Las propiedades del retículo aseguran que el método recupera los mejores casos (según el criterio de mayor satisfacción de los objetivos pedidos) y en orden por lo que el proceso termina en cuanto encuentre algún caso:

- `Goal_Paso1`: recupera casos que satisfagan el máximo número (todos si es posible) de objetivos de la consulta y ninguno superfluo.

$$\bigcup_{i=1}^{n} \text{GTO}(\text{Oi}: \text{level}\,0: \text{direct true})$$

- `Goal_Paso2`: recupera casos que satisfagan el máximo número (todos si es posible) de objetivos de la consulta aunque satisfaga también algún objetivo adicional (no pedido). Para ello recorre los subconceptos de cada Oi, nivel por nivel hacia abajo en la jerarquía, es decir, hacia conceptos más *específicos*. De esta forma obtenemos casos que satisfacen todos los objetivos pedidos, minimizando el número de objetivos superfluos.

$$\bigcup_{j=1}^{depth(bot)-depth(Oi)} \bigcup_{i=1}^{n} \text{GTO}(\text{Oi}: \text{level j}: \text{direct true})$$

- `Goal_Paso3`: recupera casos que satisfagan el máximo número de objetivos pedidos pero no todos. Para ello recorre los superconceptos de cada Oi nivel por nivel, hacia conceptos más generales para obtener casos minimizando el número de objetivos no cumplidos.

$$\bigcup_{j=-1}^{-depth(Oi)} \bigcup_{i=1}^{n} \text{GTO}(\text{Oi}: \text{level j}: \text{direct true})$$

El método llevará a cabo los tres pasos secuencialmente y cada uno sólo se lleva a cabo si el anterior no recupera ningún caso. Los siguientes ejemplos ilustran el método de recuperación sobre el retículo de objetivos de la Figura 5-16 (derecha). Aunque se muestran los tres pasos el proceso pararía en cuanto se recupere algún caso.

■ ***Ejemplo 1:***

`query_goals: A, B, C`

Clasificar `query_goals`: `(get-types query_goals) = O5`

- `Goal_Paso1: GTO(O5, 0) = (get-direct-instances O5)= Goal1`

Recuperamos `Caso1` como el único caso que satisface exactamente los requisitos pedidos.

- `Goal_Paso2: depth(BOT)=4 ; depth(O5)=3;`

  `GTO(O5, 1) = (get-direct-instances BOT)= ∅`

No hay más casos que satisfagan todos los objetivos pedidos.

- `Goal_Paso3: GTO(O5, -1) ∪ GTO(O5, -2) ∪ GTO(O5, -3)=`

`(get-direct-instances O3) ∪ (get-direct-instances O2) ∪`
`(get-direct-instances O1) ∪ (get-direct-instances TOP)= Goal2, Goal6, Goal4`

Recuperamos Caso2, Caso6 y Caso4 que satisfacen algunos de los objetivos pedidos (A,B,C), porque no hay ningún caso que los cumpla todos. En concreto, Caso2 y Caso6 satisfacen los objetivos A y C, y Caso4 satisface A y B (ver Figura 5-16).

Los siguientes ejemplos ilustran, además del método de recuperación, la formulación de consultas, como un proceso incremental en el que el usuario proporciona ciertos objetivos, y el sistema propone otros usando las reglas de dependencias.

- ***Ejemplo 2:***

`query_goals: B`

Usando la regla de dependencia B→ A, el sistema sugiere completar la consulta añadiendo el objetivo A. Si el usuario no acepta la sugerencia, el proceso de recuperación no realiza ninguna búsqueda porque las propiedades del retículo aseguran que no hay casos que cumplan B y no cumplan A. Si el usuario acepta la sugerencia, el método trabaja con la consulta `query_goals`: A, B y recupera casos similares usando clasificación de la siguiente forma:

Clasificar `query_goals`: `(get-types query_goals) = O2`

- `Goal_Paso1: GTO(O2, 0) = (get-direct-instances O2)= Goal4`

Recuperamos `Caso4` como el único caso que satisface exactamente el objetivo pedido: B.

- `Goal_Paso2: depth(BOT)=4 ; depth(O2)=2; GTO(O2, 1) ∪ GTO(O2, 2)=`

`(get-direct-instances O5) ∪ (get-direct-instances O6) ∪`
`(get-direct-instances BOT)= Goal1, Goal3`

Recupera `Caso1` y `Caso3` que satisfacen todos los objetivos pedidos aunque también cumplen otros objetivos no pedidos y posiblemente necesitarán adaptación.

- `Goal_Paso3: GTO(O2, -1) ∪ GTO(O2, -2) =`
  `(get-direct-instances O1) ∪ (get-direct-instances TOP)= ∅`

Recuperación sobre el retículo de precondiciones

El método de recuperación sobre el retículo de precondiciones –representado por el individuo canónico `irecognizePre_Method`– basa la recuperación en el retículo de precondiciones e intentará encontrar casos aplicables, es decir, cuya precondición se satisface en la situación descrita por la consulta. El método obtiene la clasificación del individuo `query_desc` y obtiene un conjunto de conceptos {P1,..,Pn} que pertenecen al retículo de precondiciones. En el siguiente paso recupera las instancias directas de todos los Pi, lo que corresponde a una recuperación exacta. En caso de que la recuperación exacta no genere resultados, es decir, los conceptos Pi no tienen instancias directas, la recuperación aproximada encuentra, en primer lugar, otros casos aplicables y, en segundo lugar, otros casos que sin ser totalmente aplicables, maximicen el número de propiedades de su precondición que la situación descrita en la consulta satisface. Las propiedades del retículo aseguran que el método recupera casos aplicables, si existen, y si no aquellos casos que comparten el mayor número de propiedades con la consulta.

- `Pre_Paso1`: intenta recuperar casos cuya precondición encaja exactamente con la situación descrita en la consulta.

$$\bigcup_{i=1}^{n} \mathrm{GTO}(\mathrm{Pi}:\mathrm{level0}:\mathrm{direct\ true})$$

- `Pre_Paso2`: recupera otros casos aplicables, es decir, casos cuyas propiedades en la precondición son un subconjunto de las propiedades dadas en la consulta. Para ello recorre los *superconceptos* de cada Pi, nivel por nivel hacia conceptos más genéricos. De esta forma obtenemos de forma ordenada los casos que maximizan el número de propiedades satisfechas.

$$\bigcup_{j=-1}^{-depth(Pi)} \bigcup_{i=1}^{n} GTO(Pi: level\ j: direct\ true)$$

  Este método recorre el retículo de precondiciones de forma dual al recorrido sobre el retículo de objetivos. Después de intentar la recuperación exacta, en el retículo de precondiciones primero se generalizan los conceptos Pi (recorrido hacia arriba), para encontrar aquellos casos con las precondiciones más débiles que pudieran ser aplicables en la situación consulta. Sin embargo, en el retículo de objetivos, primero se especializan los conceptos Oi (recorrido hacia abajo) para encontrar casos que satisfagan objetivos adicionales además de los pedidos en la consulta.
- `Pre_Paso3`: recupera casos no aplicables, es decir, cuya precondición no se satisface en la situación de la consulta. Para ello recorre los subconceptos de cada Pi nivel por nivel para obtener casos que minimicen el número de propiedades de su precondición que no se satisfacen.

$$\bigcup_{j=1}^{depth(bot)-depth(Pi)} \bigcup_{i=1}^{n} GTO(Oi: level\ j: direct\ true)$$

El método llevará a cabo los tres pasos secuencialmente y cada uno sólo se lleva a cabo si el anterior no recupera ningún caso. Los siguientes ejemplos ilustran el método de recuperación sobre el retículo de precondiciones de la Figura 5-16 (izquierda).

■ ***Ejemplo 3:***

En la situación consulta se satisfacen las propiedades X y Z, se construye el individuo `query_desc: X, Z`

Reglas de dependencia aplicables: X→Y

El sistema pregunta si la situación también satisface la propiedad Y, ya que todos los casos que incluyen X, Z en su precondición, también incluyen Y.

Clasificar `query_desc`: `(get-types query_desc) = P4`

- `Pre_Paso1: GTO(P4 :level 0)=(get-direct-instances P4)= Pre_2`

Recuperamos `Caso2` como el caso mejor porque su precondición encaja exactamente con la situación descrita en la consulta.

- `Pre_Paso2: depth(P4)=3; GTO(P4, -1)∪GTO(P4, -2)∪GTO(P4,-3)= (get-direct-instances P5) ∪ (get-direct-instances P1) ∪ (get-direct-instances P3) ∪ (get-direct-instances P2) ∪ (get-direct-instances TOP)= Pre4, Pre1, Pre3`

Recuperamos `Caso4`, `Caso1` y `Caso3` que son casos aplicables, es decir, la situación consulta satisface su precondición.

- `Pre_Paso3: depth(BOT)=4; depth(P4)=3;`

`GTO(P4 :level 1) = (get-direct-instances P6) = Pre_5`

Recupera `Caso5` que, aunque no es aplicable y requerirá adaptación, comparte propiedades con la situación consulta.

Recuperación sobre ambos retículos

El método `irecognizeGoalPre_Method` basa la recuperación en ambos retículos simultáneamente. Este método recuperará, cuando sea posible, casos aplicables (es decir, casos cuya precondición se satisface en la situación consulta) que satisfagan todos los objetivos pedidos. Las propiedades de los retículos permiten asegurar que estos casos, si existe alguno, se recuperan como las instancias directas de la clasificación en ambos retículos (`Goal_Paso1` y `Pre_Paso1`) e intersecando los resultados.

Cuando esta intersección sea vacía, el proceso intentará la recuperación aproximada: encontrar casos no totalmente aplicables aunque cumplen todos los objetivos pedidos, o casos aplicables aunque no satisfacen todos los objetivos de la consulta. Se definen dos estrategias:

- La estrategia 1 considera que es mejor recuperar casos que satisfagan todos los objetivos pedidos aunque no sean totalmente aplicables, que casos aplicables que no cumplan todos los objetivos. Esta estrategia fija la clasificación en el retículo de objetivos y recorre el retículo de precondiciones, buscando primero entre los casos aplicables (pasos 1 y 2) y después entre los casos no aplicables (paso 3).

```
j =1; i=1; Casos_Recuperados = ∅;
While (Casos_Recuperados = ∅) and j<4
     While (Casos_Recuperados = ∅) and i<4
     begin
     Casos_Recuperados=Casos_Recuperados∪(Pre_Pasoi∩Goal_Pasoj);
     i=i+1; j=j+1;
     end;
```

- Las estrategia 2 considera que es mejor recuperar casos aplicables aunque no satisfagan todos los objetivos pedidos.

```
j =1; i=1; Casos_Recuperados = ∅;
While (Casos_Recuperados = ∅) and i<4
     While (Casos_Recuperados = ∅) and j<4
     begin
     Casos_Recuperados=Casos_Recuperados∪(Pre_Pasoi∩Goal_Pasoj);
     i=i+1; j=j+1;
     end;
```

Cada una de las estrategias define un orden con el que recorrer completamente los dos retículos. Los siguientes ejemplos ilustran el proceso:

■ ***Ejemplo 4:***

`query_desc: X, Y, Z`

`query_goals: A B C`

Estrategia 1:

$Pre\_Paso1 \cap Goal\_Paso1 = \{Caso2\} \cap \{Caso1\} = \emptyset$

$Pre\_Paso2 \cap Goal\_Paso1 = \{Caso1, Caso4, Caso3\} \cap \{Caso1\} = \{Caso1\}$

Se recupera `Caso1` como el caso mejor, es decir, es aplicable y satisface todos los objetivos pedidos.

Estrategia 2:

$Pre\_Paso1 \cap Goal\_Paso1 = \emptyset$

`Pre_Paso1 ∩ Goal_Paso2 = ∅`

`Pre_Paso1 ∩ Goal_Paso2 = Caso2`

Se recupera `Caso2` como el caso mejor ya que su precondición encaja exactamente con la situación de la consulta y satisface algunos de los objetivos pedidos.

- ***Ejemplo 5:***

`query_desc:` `T`, usando la dependencia (T → Y) se completa a: T, Y

`query_goals: A, C`

Ambas estrategias recuperan `Caso6` como el caso mejor, porque encaja exactamente con la consulta.

`Pre_Paso ∩ Goal_Paso1 = {Caso6} ∩ {Caso2, Caso6} = {Caso6}`

Este método de recuperación basado en los retículos de objetivos y precondiciones aprovecha la clasificación, es eficiente, está bien fundamentado formalmente y utiliza conocimiento de la base de casos para hacer las búsquedas.

Trabajo relacionado con la recuperación por precondiciones y objetivos

El método de recuperación por precondiciones y objetivos se relaciona con la recuperación descrita en los trabajos de J. Koelher [Koelher94] y A. Napoli [Napoli&Lieber96] que también consideran la recuperación de casos como un proceso de clasificación del índice de la consulta en una jerarquía permitiendo localizaciones flexibles que recorren ascendente o descendentemente la jerarquía de precondiciones y objetivos. El método de recuperación en los retículos de precondiciones y objetivos utilizando el operador GTO comparte las ideas propuestas en los trabajos anteriores, siendo la diferencia fundamental precisamente la estructura de recuperación, es decir, los retículos de precondiciones y objetivos construidos a través de la aplicación del AFC a los casos. En los trabajos de J. Koelher y A. Napoli, se presupone la construcción *manual* de una estructura conceptual adecuada sobre la que se organizan los casos y trabaja el proceso de recuperación descrito. Nuestra aproximación evita el esfuerzo del diseñador que no tiene que definir los conceptos que representan agrupaciones interesantes de objetivos y precondiciones, porque el proceso de AFC se encarga de ello.

Este método también está relacionado con el sistema de planificación PRODIGY/ANALOGY [Veloso94] donde el proceso de recuperación busca casos que satisfagan todos los objetivos planteados por el problema consulta y cuyas *huellas* encajen con la situación inicial de la consulta (*footprint-based retrieval*) [Veloso94] [Muñoz&Hüllen95] [Muñoz&Hüllen96]. Se definen las huellas de un objetivo como las características del estado inicial que contribuyen a conseguir ese objetivo. Esta idea está relacionada con el uso que hacemos de las precondiciones. En la primera fase de la recuperación se busca un caso que satisfaga todos los objetivos pedidos en la consulta. Si ocurre un fallo, en PRODIGY/ANALOGY se buscan casos divididos en dos grupos: casos que satisfacen todos los objetivos menos uno y casos que satisfacen el objetivo restante. Si esto no genera resultados, el proceso de descomposición continua hasta que se intenta satisfacer cada objetivo independientemente del resto. En relación con nuestro trabajo, podemos resaltar que el método de recuperación sobre el retículo de objetivos satisface los mismos requisitos que el método propuesto en PRODIGY/ANALOGY utilizando mecanismos de clasificación más eficientes.

En CAPLAN/CBC [Muñoz&Hüllen95] se intenta solventar el alto coste del mecanismo de encaje entre conjuntos de objetivos propuesto en PRODIGY/ANALOGY. En la arquitectura del sistema CAPLAN/CBC se consideran explícitamente las dependencias estructurales entre

objetivos. Trabajan con dominios para los que las descripciones de los problemas incluyen dependencias entre los objetivos, que pueden determinarse antes del comienzo del proceso de planificación. Un ejemplo de dependencias es el orden entre los pasos que consiguen un conjunto de objetivos, lo que establece también un orden parcial entre los objetivos. Las restricciones en el orden de consecución de objetivos se usan durante la recuperación como una restricción adicional que deben satisfacer los casos recuperados.

En nuestra propuesta, los conceptos formales de los retículos de precondiciones y objetivos simplifican el proceso de encaje entre conjuntos de objetivos y precondiciones usando la clasificación de conceptos como un mecanismo muy eficiente. A diferencia de CAPLAN/CBC las dependencias entre objetivos no son proporcionadas explícitamente como parte de la descripción de la consulta, sino que se extraen de la base de casos. En nuestra aproximación, el conocimiento de dependencias se refiere a la co-aparición de objetivos y características en los casos y no tiene en cuenta el orden (lo que puede resultar un inconveniente). La organización de CAPLAN/CBC no es adecuada en dominios para los que no se hayan definido las dependencias entre objetivos, sin embargo nuestra aproximación usando AFC es independiente del dominio y de la base de casos.

Otro inconveniente de nuestra aproximación, que resolveremos en un futuro próximo, es que no permite considerar que unos objetivos o propiedades tengan más importancia que otras. El proceso maximiza cuantitativamente el número de objetivos o propiedades que se satisfacen, pero no maximiza cualitativamente la utilidad del caso recuperado. Sin embargo, en los sistemas de planificación es usual que distintas características contribuyan de forma distinta a la solución. Pretendemos incorporar las ideas propuestas en [Muñoz&Hüllen96] para refinar su métrica anterior añadiendo pesos a las características (*weighted footprint-based retrieval*). Los pesos se recomputan en función del rendimiento de los casos durante los episodios de resolución de problemas.

3.3 Recuperación por criterios de relevancia

En [Ashley&Aleven93] se describe un sistema tutor (CATO) para enseñar a los alumnos de un curso de leyes a razonar con casos. Para ello representa los criterios que miden la relevancia de los casos en lógica de primer orden utilizando LOOM. En CBROnto hemos definido un método de recuperación que se basa en la formulación original de Kevin Ashley y en aprovechar la capacidad que ofrece el sistema LOOM para definir consultas utilizando expresiones en lógica de predicados de primer orden.

El intérprete de consultas de LOOM permite utilizar una consulta en lógica para recuperar información de una base de conocimiento. En CBROnto hemos incluido un método de recuperación de casos basado en esta idea. Además, estas consultas pueden hacer referencia a *criterios de relevancia* que el diseñador haya definido previamente.

Un criterio de relevancia se define como el criterio en el que se basa un sistema CBR para decir que un caso es relevante para un problema y más relevante que el resto de los casos. En cualquier sistema CBR, de forma implícita o explícita, la recuperación de casos operacionaliza un criterio de relevancia. La recuperación de casos consiste en encontrar los casos que satisfacen un cierto criterio de relevancia dado –que puede estar fijado a priori o definido por el usuario en cada interacción.

En este método expresaremos los criterios de relevancia mediante una relación LOOM definida a través de una cláusula `:satisfies` (ver Apéndice A) que describirá las propiedades que satisfacen las tuplas de la relación. Una vez que hemos definido ciertos criterios de rele-

vancia como relaciones, el método de recuperación se encarga de encontrar los individuos que forman tuplas válidas de la relación, es decir, del criterio de relevancia. Realmente, en este método aprovechamos las capacidades del interprete de consultas de LOOM que se encarga de encontrar los casos (o combinaciones de casos) que satisfacen la definición.

En CBROnto existen ciertos criterios predefinidos que se pueden utilizar durante la recuperación de casos. Estos criterios utilizan terminología de CBROnto y aprovechan la integración por clasificación de los términos del dominio bajo los términos de CBROnto. Por ejemplo el siguiente criterio genérico permitirá determinar, cuando se use en una consulta, qué caso tiene más *propiedades* en común con una situación dada:

```
(defrelation more-on-point
   :constraints (domains Case Case)
   :range Case
   :is (:satisfies (?c1 ?c2 ?cfs)
               (:and   (Case ?c1)(Case ?c2)(Case ?cfs)
                       (neq ?c1 ?cfs)
                       (:for-all ?p (:implies
                                       (:and (description-property ?p)
                                          (shared-property ?c2 ?cfs ?p))
                                       (applicable-property ?c1 ?p)))
                       (:for-some ?p   (:and (Description-Property ?p)
                                       (shared-property ?c1 ?cfs ?p)
                       (:not (applicable-property ?c2 ?p))))))
```

Una tupla (c1,c2, q) pertenece a la relación si c1, c2 y q son instancias del concepto CASE de CBROnto, y el caso c1 tiene más *propiedades* en común que el caso c2 con el caso consulta q. Es decir, si c1 tiene todas las propiedades que c2 comparte con q y además, tiene alguna propiedad común adicional, que no tiene c2, con la consulta q. Las propiedades serán todas las relaciones que estén clasificadas bajo la relación `description-property` de CBROnto.

Basándose en el criterio anterior se puede definir la relación `most-on-point` de forma que la tupla (c, q) pertenece a esta relación si c es el caso que tiene más propiedades en común con la consulta q, es decir, no existe otro caso c1 con más propiedades en común que c.

Además de los criterios genéricos predefinidos, al diseñar una nueva aplicación se pueden definir criterios nuevos, utilizando terminología específica del dominio. En [Ashley&Aleven93] definen varios criterios de relevancia complejos aplicados al sistema CATO.

Supuesto que el usuario haya elegido alguno de los criterios de relevancia predefinidos, o definido uno propio, `CR`, el *método de recuperación por criterios de relevancia* recuperará casos usando el intérprete de consultas de LOOM: `(retrieve ?c (CR ?c query))`

Existen numerosas ventajas de la representación explícita y declarativa de los criterios de relevancia utilizados para la recuperación de casos. Por ejemplo, una ventaja es que permite que el usuario final decida el criterio de relevancia utilizado en cada interacción, lo que proporciona una mayor versatilidad del sistema diseñado, en el que existen distintos modos de uso del sistema con objetivos diferentes. Además, la representación declarativa facilita la implementación de múltiples criterios de relevancia y el prototipado de nuevos criterios, o la modificación de los anteriores. La representación lógica permite expresar fácilmente condiciones complejas que involucren un número arbitrario de casos interrelacionados por relaciones múltiples. Además es muy flexible: permite fácilmente formular restricciones más o menos fuertes eliminando, añadiendo o modificando ciertas condiciones. Por último, la definición simbólica utilizando una representación lógica hace más sencilla la explicación a un usuario de la elección de ese caso frente a otros basándose en un cierto criterio.

3.3.1 Resolución de las subtareas y requisitos del método

El individuo `iRetrieve_RelCriteria_Method` es la instancia canónica que representa al método de recuperación por criterios de relevancia. Es un método de descomposición que, dado un individuo consulta, divide la tarea de recuperación en dos subtareas: aplicar el criterio de relevancia dado como parámetro para recuperar los casos similares a la consulta y seleccionar el/los casos a recuperar: `AssessSim_Task` (a través de su instancia `iretrieve_relevance_task`) y `Select_Task` (a través de su instancia `iselect_case`).

La subtarea `iretrieve_relevance_task` tiene asociado (enlace `task_method`) con `iretrieve_relevance_method` el único método que tiene como competencia el individuo que representa esta subtarea. Es un método de resolución que valora la similitud de los casos utilizando el intérprete de consultas de LOOM. Para ello genera una consulta, para que dicho intérprete la resuelva, utilizando el criterio de relevancia que recibe como parámetro. Genera como salida el individuo `iretrieve_relevance_output` que contiene la lista de casos candidatos en el atributo `caseList`.

Volvemos a hacer hincapié en el hecho de que el uso de la instancia `iselect_case` para representar a la tarea de selección, permite encadenar distintos métodos de recuperación. En concreto permite configurar procesos de recuperación en dos fases, en la primera se valora la similitud usando cualquiera de los métodos disponibles, y en la segunda se selecciona usando otros métodos, como el cómputo de similitud, que sólo se aplica al conjunto de casos reducido resultante de la tarea de valoración de la similitud.

Respecto a los requisitos, no se incluyen requisitos de aplicabilidad adicionales ya que como el usuario puede definir criterios de relevancia externamente no es requisito indispensable que los criterios estén definidos previamente. Como requisito paramétrico el usuario puede indicar el criterio de relevancia concreto que se utilizará en la recuperación de entre todos los que el diseñador haya especificado como requisito de diseño. Los criterios de relevancia se construyen como relaciones que especializan a la relación `RelevanceCriteria` de CBROnto. Como requisito de conocimiento se recomienda la existencia de criterios de relevancia predefinidos (subrelaciones de `relevanceCriteria`). La definición de criterios previos es recomendable ya que permite que el diseñador ofrezca distintos modos de uso del sistema mediante los distintos criterios.

Como requisitos de diseño el método recibe:

- En el atributo `relcri_list` una lista de los criterios que se utilizarán en una interacción concreta según indique el atributo siguiente.
- En el atributo `used_criteria` el valor `order`, para indicar que los criterios se prueban en orden hasta que alguno encuentre candidatos, o el valor `user` para indicar que será el usuario el que elegirá uno de los criterios de la lista.

3.4 Recuperación por términos de similitud

Otro método de recuperación de casos que hemos incluido en CBROnto, está relacionado con algunos trabajos que hemos descrito en el Capítulo 2, en concreto [Plaza95] y [Salotti&Ventos98], que utilizan un método basado en clasificación para valorar la similitud entre casos y recuperar el caso más similar. Este método de recuperación se basa en computar los términos de similitud como el LCS entre dos conceptos que representan a las instancias, y ordenarlos en un semiretículo usando la relación de subsunción. El objetivo planteado utilizar criterios de *selección* de casos simbólicos, explícitos, formales y homogéneos.

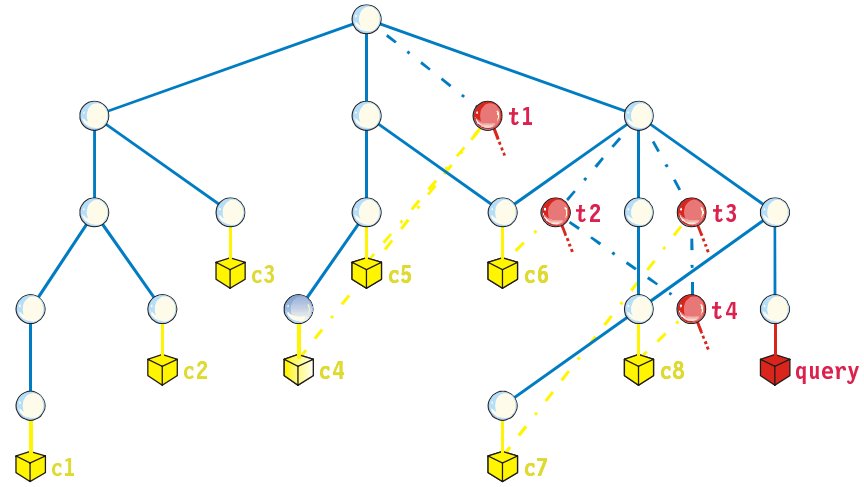


Figura 5-18. *Términos de similitud*

El método se formaliza utilizando la clasificación automática de conceptos de las DLs. En concreto se computan y representan, de manera explícita, los conceptos que representan la similitud entre los casos y sus diferencias, y se utiliza el orden parcial inducido por la relación de subsunción sobre dichos conceptos para seleccionar los casos más similares al caso consulta (Figura 5-18).

Los siguientes pasos esquematizan el *método de recuperación por términos de similitud*:

1. Suponiendo que existe un individuo `query` que representa la consulta actual, se computa el LCS entre dicho individuo y cada uno los casos. Los conceptos obtenidos se ordenan automáticamente en una taxonomía. Llamaremos E_{LCS} al conjunto de los conceptos más específicos de esta taxonomía.
2. Cada concepto de E_{LCS} es un término (concepto) de similitud con la consulta que se corresponde con uno o más casos, que serán recuperados como los más similares a la consulta. Estos casos son más similares que el resto porque su concepto de similitud con la consulta es más específico que el de otros casos.

Una mejora de este método de recuperación, que incluiremos en CBROnto en un futuro próximo, se basa en la extensión a la aproximación original de E. Plaza descrita en [Salotti&Ventos98]. En este trabajo, además de los conceptos de similitud proponen computar conceptos que representan las diferencias entre los casos. De esta forma se puede hacer una criba inicial eligiendo los casos asociados al término de similitud más específico, y usar los términos de diferencia (*dissimilarity terms*) para seleccionar entre todos los casos que compartan ese término de similitud con la consulta dada. Para ello también clasifica los términos de diferencia usando la relación de subsunción y selecciona los casos asociados al término de diferencia menos específico. Es decir, los casos más similares.

3.4.1 Resolución de las subtareas y requisitos del método

El método de recuperación por términos de similitud –representado por el individuo canónico `iRetrieve_SimilarityTerms_Method`– no incluye requisitos adicionales Es un método de descomposición que, dado un individuo consulta, divide la tarea de recuperación en tres subtareas: filtrar la base de casos inicial para obtener un conjunto de casos reducido, valorar

la similitud computando los términos (conceptos) de similitud entre la consulta y cada uno de los casos de este conjunto. Por último, seleccionar los casos mejores que serán los casos asociados al término de similitud más específico de todos los que se han generado: `Obtain-Cases_Task` (a través de su instancia `iFilter_Cases`), `AssessSim_Task` (a través de su instancia `iCompute_simTerms_Task`) y `Select_Task` (a través de su instancia `iselect_case`).

La subtarea `iFilter_Cases` tiene como objetivo obtener el conjunto de casos al que se aplicarán las siguientes subtareas. Existen varios métodos cuya competencia es adecuada para resolver esta tarea:

- El método `iretrieve_relevance_method` es un método de resolución que valora la similitud de los casos utilizando el intérprete de consultas de LOOM. De esta forma, filtra los casos sobre los que se computarán los términos de similitud. Este método se describió en el Apartado 3.3 ya que también tiene como competencia la subtarea `iretrieve_relevance_task` generada por el método de recuperación por criterios de relevancia.
- El método `iobtain_case_base` que obtiene todas las instancias de la base de casos que recibe especificada en el parámetro `case_base` (heredado). El método genera como salida un individuo (`iobtain_cases_output`) con un atributo `caseList` que lo relaciona con el conjunto de casos de salida. Este método se describió en el Apartado 3.1.2 ya que también tiene como competencia la subtarea `iobtain_cases` que es la primera de las subtareas del método de recuperación por cómputo de similitud.
- El método `inumeric_simcomputation_method` que se encarga de computar la similitud entre la consulta (parámetro `query`) y cada uno de los casos del conjunto generado por el método que resuelve la subtarea anterior (requisito de secuencia `caseList`) o de la base de casos (parámetro `casebase`). Este método también fue descrito en el Apartado 3.1.2 ya que también tiene como competencia la subtarea `icompute_similarity` que es la segunda de las subtareas que genera el método de recuperación por cómputo de similitud.

El uso de métodos con varias competencias, permite encadenarlos en las secuencias de resolución de distintas formas. Por ejemplo, el método `iretrieve_relevance_method` puede ser el primero de la secuencia de tareas generada por el método `iRetrieve_RelCriteria_Method` donde tras el uso de un criterio de relevancia se selecciona uno de los casos, o el primero de la secuencia de tareas generada por el método `iRetrieve_SimilarityTerms_Method` donde tras el filtrado inicial usando un criterio de relevancia, se computan los términos de similitud y después se selecciona uno de los casos candidatos después de la aplicación secuencial de los dos métodos.

La subtarea `iCompute_simTerms_Task` está asociada (enlace `task_method`) al método `iCompute_simTerms_Method` que se encarga de computar los términos de similitud entre la consulta y cada uno de los casos en la lista `caseList` generada por los métodos anteriores. Después de clasificar y elegir los casos candidatos se devuelven como salida en el atributo caseList del individuo `iCompute_simTerms_output`. La tarea `iselect_case` –descrita en el Apartado 3.2.1.2– se encarga de la selección de uno de estos casos candidatos.

4. Adaptación

Después de resolver la tarea de recuperación, el objetivo de la tarea de reutilización o adaptación de casos (REUSE_TASK) es adaptar el mejor caso recuperado para que satisfaga los requisitos especificados por la consulta.

Los métodos de adaptación de CBROnto se basan en hacer ciertas transformaciones sobre el caso recuperado, por ejemplo, añadir, eliminar o sustituir alguno de sus elementos. Aunque este tipo de métodos encajarían más con una aproximación transformacional al CBR, también se aplica en aproximaciones derivacionales donde las soluciones almacenan una traza de los pasos de razonamiento que se han utilizado para resolver el problema. El método de adaptación consistirá en la aplicación de la misma traza a la descripción de la consulta: se realiza una copia de la traza que se devuelve como la solución de la consulta y se sustituyen los objetos sobre los que se llevarán a cabo las acciones de la solución. En concreto, si una acción de la solución hace referencia a un elemento E1 de la descripción del caso, y en la consulta no aparece el elemento E1, pero en la misma posición, es decir, siguiendo el camino de relaciones que conecta el caso con E1, aparece un elemento E2, se sustituyen las referencias a E1 por referencias a E2. En el caso de que el elemento E2 no sea único se propone el elemento más similar a E1, respecto a los conceptos a los que pertenece y sus relaciones con otros individuos, pero será el usuario el encargado de validar estas sustituciones.

Como hemos descrito en el Capítulo 4, en CBROnto existe un método que resuelve la tarea de adaptación de casos descomponiéndola en dos subtareas. `Copy_Solution_Task` cuyo objetivo es hacer una copia sobre el individuo consulta de la componente de solución del caso recuperado. Además como el caso recuperado no es único en general, el método de adaptación elegirá el primero (ya que están ordenados por similitud) a menos que tenga un resultado de fallo, en cuyo caso se le preguntará al usuario si quiere utilizarlo. La siguiente subtarea, `Adapt_Solution_Task`, tiene como objetivo la modificación de dicha solución. Para resolver esta última tarea existen varios métodos alternativos que describimos en este apartado. La salida del método `iReuse_Method` está representada por el individuo `iReuse_Output` una instancia de `Method_Output` que incluye los atributos `reuse-result`, que puede tener como valores `success` o `failure`, para indicar el éxito del método de adaptación, y `adaptedCase` que hace referencia al caso adaptado. Este caso sólo es correcto para resultados de éxito (`success`) del método de adaptación.

Los métodos de adaptación de la solución (`Adapt_Solution_Method`) incluidos en CBROnto se basan en la aplicación de ciertas transformaciones estructurales genéricas complementadas con el uso de estrategias de adaptación específicas del dominio. Según varios autores [Carbonell83] [Kolodner93] un pequeño conjunto de transformaciones es suficiente para caracterizar un amplio rango de adaptaciones, a costa de requerir mucho conocimiento específico del dominio para encontrar la información necesaria para aplicar estas transformaciones. En los trabajos de David Leake en el marco del sistema DIAL [Leake95 a y b] [Leake *et al.* 96] [Leake *et al.* 97 a y b] la búsqueda de esta información se basa en un proceso de generación de objetivos de conocimiento y su uso para buscar información y empaquetar la traza de razonamiento que puede guiar procesos futuros de adaptación. Los objetivos de conocimiento se representan típicamente en dos partes: especificación de un concepto que proporciona una plantilla con la que debe encajar la información (el candidato) que estamos buscando, y una descripción de cómo usar esa información una vez que la hayamos encontrado. Leake añade además una especificación comparativa que describe cómo escoger entre múltiples alternativas que satisfagan la especificación dada por el concepto; y también incluye información sobre la cantidad de esfuerzo que se permite para intentar satisfacer este objeti-

vo de conocimiento, medido en términos del número máximo de operaciones primitivas de acceso a memoria que deben ser aplicadas durante el proceso de búsqueda.

El siguiente paso es buscar en la base de conocimiento la información necesaria para aplicar la transformación elegida (por ejemplo, encontrar un sustituto adecuado). Leake usa *razonamiento introspectivo* [Leake95b] [Leake *et al.* 95] [Fox&Leake95] para encontrar posibles estrategias de búsqueda de información en memoria. Su proceso de razonamiento implementa la búsqueda en memoria como una forma de planificación, usando operadores que describen acciones realizadas sobre la memoria. El uso de un proceso de planificación facilita una recombinación flexible del conocimiento de búsqueda en memoria (se puede por ejemplo, concatenar un plan después de otro). Las estrategias de búsqueda también se aprenden (en forma de *casos de búsqueda en memoria*). Se genera un plan de búsqueda en memoria que puede incluir tanto los operadores iniciales de *estrategias de búsqueda en memoria* o operadores de aplicación de casos de búsqueda en memoria almacenados previamente.

En el trabajo presentado en [Kass90] se propone un modo de afrontar el problema de generalidad/operacionalidad de las reglas de adaptación utilizadas para modificar la solución del caso. El uso de reglas de adaptación específicas del dominio que incluyan mucho conocimiento requiere un gran esfuerzo de adquisición de conocimiento y son poco reutilizables pero muy fáciles de aplicar, ya que tenemos todo el conocimiento necesario codificado en la regla. Por otra parte, las reglas abstractas son aplicables a una clase amplía de problemas de adaptación pero no dan guías sobre cómo encontrar el conocimiento específico necesario para aplicarlas. Kass propone el uso de *estrategias de adaptación* que combinan transformaciones genéricas, con estrategias de búsqueda en memoria independientes del dominio para encontrar la información específica del dominio necesaria para aplicar las estrategias. Nuestra aproximación a la adaptación utiliza este tipo de estrategias de adaptación. También se relaciona con el trabajo de David Leake si consideramos que las estrategias de adaptación se corresponden con lo que él llama *casos de adaptación*. En la propuesta de Leake los casos de adaptación representan una única transformación simple, por ejemplo, una sustitución o un elemento añadido o eliminado, que soluciona un cierto problema. Aunque relacionada, en el estado actual nuestra aproximación es mucho más simplista que la propuesta de Leake. Nuestro objetivo, no es el de proporcionar métodos de adaptación muy sofisticados sino el de dotar de una estructura común para ellos, que nos permita experimentar con métodos simples que puedan ser complementados con conocimiento específico del dominio para permitir procesos más complejos.

En CBROnto distinguimos entre un método de adaptación de la solución (en adelante simplemente método de adaptación) genérico y un método especializado, que requiere conocimiento específico de adaptación para cada aplicación CBR diseñada. El método de adaptación genérico utiliza una estrategia de adaptación predefinida que usa como único tipo de transformación la sustitución de ciertos elementos de la solución del caso, y utiliza métodos genéricos de búsqueda de sustitutos aprovechando el conocimiento del dominio [González *et al.* 99a]. Comenzamos describiendo el método de adaptación especializado que, aunque es más sofisticado y permite obtener mejores resultados, requiere la representación explícita de ciertas estrategias específicas. El Apartado 4.2 describe el método de adaptación genérico basado en sustituciones. El objetivo de este capítulo es la descripción del comportamiento de los métodos desde un punto de vista genérico, independientemente de su uso en una aplicación concreta. En el Capítulo 6 veremos el proceso de diseño de aplicaciones y cómo utilizar los métodos de adaptación descritos en este apartado.

4.1 Método de adaptación especializada basado en estrategias

El método `iAdapt_Specialized_Method` adapta la solución del caso recuperado utilizando ciertas estrategias de adaptación especificadas por el diseñador de la aplicación. Estas estrategias de adaptación representan conocimiento específico de la aplicación que indican cómo y cuándo llevar a cabo un paso de transformación sobre el caso recuperado. Las estrategias de adaptación se adquieren durante la fase de diseño del sistema y representan qué transformaciones se aplicarán y, para cada una, el método de búsqueda que vamos a utilizar para encontrar la información necesaria para aplicarla.

4.1.1 Resolución de las subtareas

El método `iAdapt_Specialized_Method` descompone la tarea de modificación de la solución en tres subtareas que se resuelven de forma cíclica:

1. Determinar los problemas de la solución y basados en ellos buscar una estrategia de adaptación adecuada para la situación actual `Select_Strategy_Task` (a través de su instancia `iretrieve_adaptation_strategy_task`).
2. Utilizando la estrategia anterior, la tarea `Select_Discrepancy_Task` (a través de su instancia `ifind_adaptation_actions_task`) identifica una lista de acciones de transformación que se llevarán a cabo sobre la solución del caso, así como los elementos involucrados en estas transformaciones.
3. La subtarea `Modify_Solution_Task` (a través de su instancia `imodify_solution_task`) se encarga de resolver la lista de discrepancias aplicando las transformaciones indicadas por la estrategia de adaptación.

Para resolver la tarea `iretrieve_adaptation_strategy_task` se utiliza el método de resolución `iretrieve_adaptation_strategy_method` que identifica los problemas que plantea el caso recuperado en la situación actual y en base a ellos, recupera las estrategias especificadas por el diseñador. Como veremos, las estrategias añaden pautas de aplicabilidad para determinar sobre qué elementos se aplicarán las transformaciones.

Existen tres métodos que resuelven la tarea `ifind_adaptation_actions_task y` que utilizan estas pautas de aplicabilidad para identificar la lista de acciones de transformación que se llevarán a cabo sobre la solución del caso, así como los elementos involucrados en estas transformaciones. El método `iuser_find_adaptation_actions_Method` delega la resolución de la tarea al usuario final, el método `ifix_adaptation_items_Method` aplica las transformaciones especificadas por la estrategia a ciertos elementos de la solución que ocupan una posición fija (especificada mediante una cadena de relaciones), y el método `isystem_find_adaptable_actions_method` identifica los elementos sobre los que aplicar las acciones utilizando las diferencias entre el caso y la consulta.

Para resolver la tarea `imodify_solution_task` se utiliza un método `imodify_solution_method` que de forma cíclica, resuelve las dos subtareas siguientes:

- `iapply_transformation_task` a través del método `apply_transformation_method`, encargado de aplicar una a una las transformaciones de la lista de acciones que resultan de la tarea anterior.
- `ilocal_revisión_task` que se encarga de validar o rechazar cada uno de los pasos de transformación de la solución. Además del método `iDo_Nothing_Method` (ya que la tarea no es obligatoria) existe un único método cuya competencia es adecuada pa-

ra resolver esta tarea: iuser_local_revisión_method que delega al usuario la validación de cada paso de transformación.

El método imodify_solution_method genera como salida el individuo iReuse_Output (salida del método de adaptación de casos) que incluye los atributos reuse-result, que puede tener como valores success o failure, para indicar el éxito del método de adaptación, y adaptedCase que hace referencia al caso adaptado. Se considera que el proceso de adaptación ha tenido éxito si se han llevado a cabo todas las transformaciones previstas en la estrategia y en caso de validación local, todas las transformaciones hayan sido validadas.

4.1.2 Requisitos del método

Los requisitos de aplicabilidad del método de adaptación especializada se basan en la necesidad de la definición previa de ciertas estrategias de adaptación especificadas por el diseñador de la aplicación. Además, hereda un requisito de aplicabilidad del método de adaptación de casos relativo a la existencia de solución en el caso que se adapta.

Respecto a los requisitos de entrada, no se especifican requisitos paramétricos, ni de diseño. Como parte de los requisitos de conocimiento se requiere la formalización de las estrategias de adaptación (instancias del concepto Adaptation_Strategy) y de los tipos de problemas (subconceptos de Problem_Type), como se describe en el siguiente apartado. Por último, los requisitos de secuencia (individuo iAdapt_Solution_Sequence) especifican la existencia del atributo casetoAdapt, cuyo relleno es un individuo (instancia de Case_with_Solution) que incluye la descripción de la consulta y la solución del caso recuperado, y el atributo caseList con la lista de casos recuperados.

4.1.3 Los tipos de problema y las estrategias de adaptación

Como ya hemos comentado, las estrategias de adaptación representan conocimiento específico del dominio y/o de la aplicación que indican cómo y cuándo llevar a cabo las transformaciones de la solución del caso recuperado. Las estrategias de adaptación se adquieren durante la fase de diseño del sistema, seleccionando qué transformaciones se aplicarán, sobré qué elementos y, para cada una, el método de búsqueda que vamos a utilizar para encontrar la información necesaria para aplicarla.

Durante la fase de diseño de la aplicación el diseñador definirá, en términos del dominio, cuáles son los *tipos de problemas* que requieren adaptación y la estrategia de adaptación prevista para solucionar cada uno de estos tipos de problemas. Para ello, se definen las características del tipo de problema y de la estrategia de adaptación con las que COLIBRI construirá, respectivamente, un concepto (subconcepto de Problem_Type) y una instancia del concepto Adaptation_Strategy que se anota en el concepto tipo de problema a través de la relación de CBROnto has_adaptation_strategy.

La representación de los conceptos que representan los tipos de problemas no se basa en un vocabulario genérico sino en el vocabulario específico del dominio. En concreto, el diseñador debe definir los tipos de problemas en términos de la clasificación conceptual de los individuos que representan al caso y a la consulta. Durante la resolución de la tarea iretrieve_adaptation_strategy_task, cuyo objetivo es seleccionar la estrategia que vamos a aplicar, el método iretrieve_adaptation_strategy_method construye un individuo que representa los objetivos de la consulta que el caso recuperado no satisface. Para ello crea una instancia del concepto Problem_Type y aserta sobre él aquellos conceptos a los que pertenece la

consulta y el caso no pertenece, y las cadenas completas de relaciones de la consulta que no se cumplen *exactamente* en el caso.

En función de estas características, este individuo se clasificará como una instancia de alguno de los conceptos que el diseñador habrá identificado como problemas que requieren adaptación (subconceptos de `Problem_Type`). Además, asociado con ese concepto encontraremos la estrategia de adaptación prevista por el diseñador para solucionar este problema.

Supongamos que en el ejemplo de la planificación del movimiento de bloques el diseñador identifica como tipos de problemas que una pieza no esté dentro de un contenedor de tipo `Box1` o de tipo `Box2` cuando la consulta así lo requiera.

Esto se traduce en la definición de subconceptos del concepto `Problem_Type` de CBROnto, por ejemplo:

```
(defconcept Not-InBox1 :is (:and Problem_Type InBox1))
(defconcept B-Not-InBox1 :is (:and Problem_Type BinBox1))
```

El primer tipo de problema ocurrirá cuando el caso no tiene ninguna pieza o bloque dentro de un contenedor `Box1`, y el segundo tipo ocurrirá cuando el caso no tiene ninguna pieza B dentro de un contenedor `Box1`, aunque los objetivos de la consulta así lo requieran. Debido a la jerarquía de objetivos, en la que `BinBox1` es un subconcepto de `InBox1`, y a los mecanismos de clasificación automática, el concepto `B-Not-InBox1` se clasifica como un subconcepto de `Not-InBox1`.

Supongamos que se plantea una consulta en la que los objetivos que se quieren satisfacer son `AinBox1` y `BinBox1`, y supongamos también que se recupera el caso representado en la Figura 4-16 del Capítulo 4, que cumple los objetivos `AinBox1` y `BinBox2`.

El método que busca la estrategia de adaptación a aplicar creará una instancia de `Problem-Type` y asertará la pertenencia a los conceptos a los que pertenece la consulta pero no el caso, es decir, los conceptos no satisfechos por el caso recuperado. En el ejemplo asertaríamos únicamente la pertenencia al concepto no satisfecho `BinBox1`:

```
(tell (:about ad-idx Problem-Type BinBox1)
```

El mecanismo de reconocimiento de instancias clasificará este individuo bajo el concepto más específico cuya definición satisface, es decir, como una instancia del concepto `B-Not-InBox1` (subconcepto de `Problem_Type` y de `Not-InBox1`) que el diseñador ha identificado como un tipo de problema que requiere adaptación.

Además, asociado con ese concepto a través de la relación `has_adaptation_strategy` encontraremos la estrategia de adaptación prevista por el diseñador para solucionar este tipo de problemas.

4.1.3.1 Formalización de las estrategias de adaptación

Cada estrategia de adaptación se representa como una instancia del concepto `Adaptation-Strategy` de CBROnto, e incluye toda la información que el método que la aplica necesita para transformar la solución del caso. En concreto, incluye referencias a los operadores de transformación que se pueden aplicar para adaptar el caso, y cómo encontrar los elementos necesarios para aplicar las transformaciones. La aplicación de una estrategia de adaptación consiste en aplicar las transformaciones representadas y las estrategias de búsqueda en memoria respetando las restricciones específicas de la situación actual. De nuevo, aunque en el Apéndice B se pueden consultar las definiciones correspondientes, para los lectores no familiarizados con la sintaxis de LOOM incluimos una simplificación en notación EBNF:

```
<Adaptation-strategy instance> ::= [<Transformation>]+
                                   [<Strategy-Operational-Specification>].
<Transformation> ::= transformation-spec <Transformation instance>
<Transformation instance> ::=     (<Add_Transformation instance> |
                                  <Delete_Transformation instance> |
                                  <Substitute_Transformation instance>).
<Add_Transformation instance> ::= operator <Add-operator instance>
                                   [applicability <Applicability>]
                                   add-strategy <Add-search-strategy>
                                   weigth <Number>.
<Delete_Transformation instance> ::= operator <Delete-operator instance>
                                  [applicability <Applicability> ]
                                  weigth <Number>.
<Substitute_Transformation instance> ::= operator <Substitute-operator>
                                        [applicability <Applicability> ]
                                        [tosubstitute <Relation-Path>]
                                        [search-strategy <Search-Strategy instance>]
                                        weigth <Number>.
<Add-operator instance> ::= add.
<Delete-operator instance> ::= delete.
<Substitute-operator instance> ::= substitute.
<Strategy-Operational-Specification> ::= (<Strategy-With-Method> |
                                        <Strategy-With-Lisp-Function> |
                                         <Strategy-With-Method-and-Function>).
<Strategy-With-Method> ::= has-method <CBROnto-Method-instance>
<Strategy-With-Lisp-Function> ::= lisp-function <Lisp-Function-instance>
<Strategy-With-Method-and-Function> ::=<Strategy-With-Lisp-Function>
                                        <Strategy-With-Method>.
<Add-search-strategy> ::= <Simple-Search-Strategy instance>
<Search-Strategy instance> ::= <Simple-Search-Strategy instance> |
                                  <Method-Search-Strategy instance>.
<Simple-Search-Strategy instance> ::= begin-with <starting-point>
                                  [ step <Search-Step Instance>]+
                                  [weight Number]
<starting-point> ::= current-case | current-query | <individual-name>.
<Method-Search-Strategy instance> ::= [weight Number]
                                  search-method <Search-Substitutes-Method instance>
<Applicability> ::=     ({relation-name [<concept-name>]}+ |
                        <concept-name> | <individual-name>)
<Relation-Path> ::= ({relation-name [<concept-name>]}+)
<Search-Step instance> ::= step-number <Number> primitive-step <Primitive-Step>.
<Primitive-Step> ::= <Primitive-Step instance> | (<relation-name> [<concept-name>])+
<Primitive-Step instance> ::= <role-filler> | <superrelation> | <siblings-at-level>
<role-filler> ::= role-filler. ;; la instancia role-filler tiene un atributo role que
indica el nombre del atributo cuyo relleno queremos extraer.
```

`<superrelation> ::= superrelation ;; la instancia superrelation tiene un atributo rel que indica el nombre de la relación que queremos generalizar.`

`<siblings-at-level> ::= siblings-at-level ;; la instancia siblings-at-level tiene un atributo level que indica el nivel (positivo o negativo) al que recuperar instancias cercanas.`

En líneas generales diremos que cada estrategia de adaptación se describe haciendo referencia a una o varias transformaciones, que a su vez hacen referencia a uno de los operadores de transformación, sustitución (`substitute`), inserción (`add`) o borrado (`delete`) y a una descripción de cuándo aplicaremos el operador. Además, aunque para las transformaciones de borrado no se requiere conocimiento adicional, las transformaciones de inserción o sustitución harán referencia a la estrategia de búsqueda que se utilizará para encontrar el conocimiento necesario para aplicar la transformación.

Las estrategias de búsqueda para inserción de elementos se definen en términos de las operaciones primitivas de acceso a la base de conocimiento que son necesarias para encontrar valores en la memoria. Por ejemplo, extraer el relleno de un role, generalizar una relación, encontrar las instancias hermanas de otra, o en general, las instancias *primas* a nivel n. También es importante especificar a partir de qué individuo se aplica la estrategia de búsqueda, podemos usar, por ejemplo, el individuo caso actual (mediante el individuo predefinido `current-case` que se usa para hacer referencia al caso que está siendo adaptado), el individuo consulta (`current-query`) o un individuo cualquiera del dominio.

Las estrategias de búsqueda para la sustitución de elementos, además de utilizar las operaciones primitivas de acceso a memoria, pueden hacer referencia a alguno de los métodos de recuperación de casos de CBROnto configurado de forma adecuada. Esta opción no se puede utilizar en el operador de inserción porque el método de búsqueda (como se describió en la recuperación de casos) recupera un individuo similar a un elemento consulta. En el operador de sustitución el elemento a sustituir se interpreta como la consulta del método de búsqueda, es decir, sustituiremos un elemento por otro similar a él. La definición de similitud se incluye como parte de la configuración del método ya que, dependiendo del método, se especificará como una medida numérica, un criterio de relevancia, etc.

En cada una de las transformaciones existe una componente (opcional) que describe su aplicabilidad, es decir, las condiciones que debe cumplir un elemento del caso para que sobre él se aplique esta transformación. Esta componente se puede describir como un individuo, en cuyo caso la estrategia sólo es aplicable para ese individuo; como un concepto, en cuyo caso la pertenencia de un individuo al concepto determina que esta transformación es aplicable sobre este individuo; o como un camino de relaciones a partir de un individuo, con posibles restricciones conceptuales para los individuos del camino, en cuyo caso esta transformación es aplicable para los individuos tales que, a partir de ellos se pueda seguir un camino cumpliendo las restricciones conceptuales correspondientes. Además, una transformación también puede indicar que lo que quiero sustituir no es el elemento cuya aplicabilidad se cumple sino una parte suya, especificando el atributo `tosubstitute` que incluye una cadena de relaciones que conecta un elemento con alguna de sus componentes que son las que realmente se sustituyen.

La tarea `ifind_adaptation_actions_task` es la que determina la lista de elementos concretos de la solución del caso sobre los que se aplicarán las transformaciones y la transformación concreta a realizar sobre cada uno de ellos, en función de las condiciones de aplicabilidad de las transformaciones de la estrategia.

La especificación operacional de la estrategia define el comportamiento de la misma en estrategias complejas, por ejemplo, en las que la configuración del método de búsqueda de sustitutos deba variar entre distintas interacciones, o ningún método se ajusta al comportamiento deseado. Indicará el orden y las dependencias entre las transformaciones cuando no se pretenda un orden secuencial entre ellas, y la relación entre las transformaciones y los elementos del caso que hay que adaptar. Se puede definir utilizando varias opciones:

- Mediante un individuo que represente un método de la biblioteca configurado adecuadamente. Para ello existirá una relación explícita (`has-method`) entre la estrategia y el individuo que representa el método.
- Mediante una función con código Lisp cuyo parámetro de entrada será el individuo involucrado en el problema. El sistema asociará la función a la estrategia a través de un aserto con la relación (`lisp-function`). Normalmente será conveniente definir una función Lisp para especificar un flujo de control distinto al que llevaría a cabo el mecanismo de resolución de COLIBRI. Es decir, la función Lisp define un flujo de control específico que puede hacer llamadas a métodos de la biblioteca que estén configurados previamente.
- Mediante una función con código Lisp y un método que se ejecutarán en este orden. Esta opción es adecuada cuando la configuración del método no se pueda prefijar en la fase de definición de la estrategia sino que dependa, por ejemplo, del individuo reparado. En este caso la función Lisp puede realizar una configuración distinta en cada ejecución, del método de CBROnto que resuelve la reparación.

En el ejemplo del movimiento de bloques, habíamos descrito que el método que encuentra la estrategia de adaptación a aplicar, crea y clasifica un individuo bajo el concepto `B-Not-InBox1` que el diseñador ha identificado como un tipo de problema que requiere adaptación. Asociado con ese concepto –a través de la relación `has_adaptation_strategy`– encontraremos la estrategia de adaptación prevista por el diseñador para solucionar este tipo de problema:

```
(tellm (:about adaptation-B-Not-InBox1 Adaptation_Strategy
                                          (transformation-spec t1)))
(tell (:about t1 Transformation
            (operator substitute) (applicability '(over B to Box1))
            (search-strategy s1)))
(tell (:about s1 Simple-Search-Strategy
                    (begin-with current-case) (step St1)))
(tell (:about st1 Search-Step
            (step-number 1)
            (primitive-step '(has-sequence-step Sequence-Step to Box2)))
```

Ya que el problema es que el bloque B no está en el contenedor `Box1`, necesitamos una estrategia de adaptación simple que sustituye el movimiento hecho sobre B para que el destino sea `Box1`. Esto se podría llevar a cabo a través de una transformación de sustitución o de una estrategia con dos transformaciones una de borrado del movimiento erróneo y otra de inserción del movimiento correcto sobre el bloque B. El código anterior ejemplifica la transformación de sustitución. Además, en este ejemplo para resolver la tarea `ifind_adaptation_actions_task` y encontrar la lista de elementos concretos sobre los que se aplicarán las transformaciones, utilizaremos el método `ifix_adaptable_parts` que incluye

como requisito de diseño la especificación de una lista de comprobación que determina los elementos a adaptar:

`(Case has-solution Case-Solution has-sequence-step Sequence-Step)`

Este método es adecuado en el ejemplo ya que los candidatos a ser adaptados, es decir, los pasos de la solución (instancias de `Sequence-Step`) siempre ocupan la misma posición en la estructura de representación.

En cualquier caso es la estrategia de adaptación la que decide en función de la aplicabilidad de sus transformaciones. En este ejemplo la aplicabilidad se puede determinar usando el camino que indica la transformación y que se seguirá a partir del individuo `Sequence-Step`: `(over B to Box1)`.

Esto significa que esta transformación se puede aplicar cuando el paso de la solución (instancia de `Sequence-Step`) se encargue de mover una instancia del bloque B a una instancia del contenedor `Box1`.

La estrategia de búsqueda indica mediante pasos primitivos que el sustituto lo encontraremos a partir del individuo caso siguiendo el camino con restricciones conceptuales siguiente: `(has-sequence-step Sequence-Step to Box2)`.

Este método de adaptación, a diferencia del método automático basado en sustituciones, requiere un esfuerzo importante de adquisición de conocimiento de adaptación durante la fase de diseño del sistema, ya que el diseñador deberá identificar, por un lado, los tipos de problemas y, por otro, las estrategias de adaptación asociadas. Sin embargo, una ventaja de nuestra propuesta es que el tipo de problemas se define en términos del dominio y que las estrategias que se manejan son procesos de transformación flexibles y reutilizables en muchas situaciones, por lo que el uso típico no supondrá la definición de muchos tipos de problemas con estrategias muy diferentes, sino una única estrategia de adaptación aplicable en una situación muy general.

El uso más simple de este método consiste en definir una única estrategia de adaptación aplicable a todos los casos recuperados y asociarla a un tipo de problema genérico que se define como el concepto `Query-Type` de CBROnto. Esto es adecuado, si se tiene en cuenta que `Query-Type` es un concepto al que pertenecerá la consulta pero no el caso recuperado, por lo que se identificará como una discrepancia que solucionará la estrategia de adaptación. Esta idea también se puede utilizar para definir una estrategia de adaptación por defecto que se aplicará sólo cuando no haya otra más específica (es decir, asociada a un concepto tipo de problema más específico).

En el ejemplo, en vez de definir los tipos de problemas específicos como `B-Not-InBox1` se puede definir como un único tipo de problema genérico el concepto `Query-Type` de CBROnto y la estrategia de adaptación (única) incluye varias transformaciones que serán aplicables en función de sus requisitos.

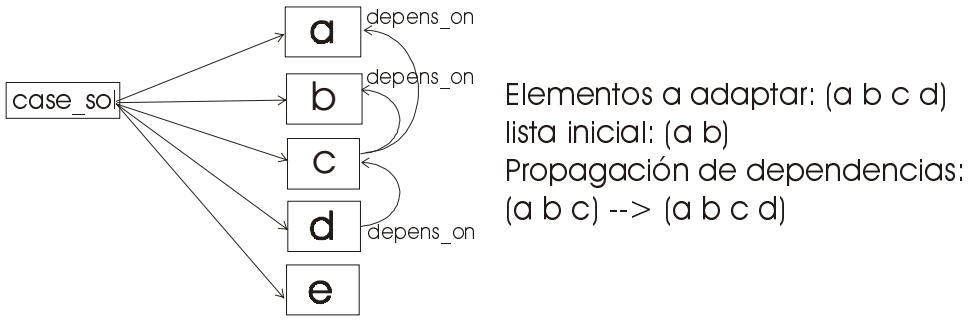


Figura 5-19. Dependencias entre elementos

4.1.3.2 Identificación de los elementos a adaptar

La subtarea `Select_Discrepancy_Task` (instancia `ifind_adaptation_actions_task`) es la segunda subtarea que genera el método de modificación de la solución. La tarea se encargará de obtener qué partes de la solución del caso no son adecuadas respecto a la consulta dada y deben ser adaptadas, así como los pasos de transformación que se llevarán a cabo sobre estos elementos. Para resolver la primera parte de esta tarea, es decir, para identificar los elementos a transformar, existen distintas alternativas que pueden estar determinadas por las diferencias entre las descripciones de los casos consulta y recuperado.

Como describimos en el Capítulo 2, si existe una representación explícita de las dependencias entre las componentes de la descripción y de la solución de un caso –o directamente la solución y la descripción comparten elementos– estas diferencias permiten obtener las partes de la solución del caso recuperado que hay que transformar. Otra alternativa para identificar las partes a adaptar, que resulta adecuada en un método de adaptación interactivo, consiste en delegar esta tarea al usuario final o al diseñador, que puede ayudar a seleccionar (al menos inicialmente) los elementos que serán objeto de alguna transformación. El diseñador puede incluir listas de comprobación de elementos predefinidas o delegar la tarea totalmente al usuario final. En todos los casos, las dependencias entre los elementos de los casos ayudan a identificar otros elementos a transformar supuesto que hemos transformado alguno del que depende.

En CBROnto las dependencias entre elementos se representan a través de la relación `depends_on`, o cualquier relación del dominio clasificada bajo la misma, que permite especificar explícitamente la dependencia entre dos elementos. Si `E1 depends_on E2`, entonces las modificaciones en `E2` suponen modificaciones en `E1`. En concreto, si `E1` y `E2` son componentes de la solución y `E2` es adaptado entonces `E1` también debe ser adaptado. Estas dependencias se utilizan durante la fase de obtención de las partes de la solución del caso recuperado que hay que adaptar, un ejemplo de la cuál se muestra en la Figura 5-19. Se puede observar que las dependencias de este tipo son de distinta naturaleza y no se corresponden con las dependencias extraídas de la base de casos durante la aplicación del AFC. Las dependencias de adaptación se representan explícitamente como parte del modelo del dominio utilizando el mecanismo de integración por clasificación en la jerarquía de relaciones.

El mecanismo que propaga las dependencias es común a los distintos métodos de identificación de los elementos a adaptar, que sólo varían en el modo de obtener la lista inicial de elementos a modificar:

- `iuser_find_adaptation_actions_Method`, se basa en la interacción con el usuario final. Este método es adecuado en sistemas con una mayor intervención del usuario.

- `ifix_adaptation_items_Method`, se basa en la recuperación de las hojas de una cadena de relaciones que se fija en la fase de diseño del sistema. Es decir, el método incluye como requisito de diseño la especificación de una lista de comprobación que determina los elementos a adaptar. Este método es adecuado si los candidatos a ser adaptados siempre ocupan la misma posición en la estructura de representación. El método permite otro requisito de diseño (atributo `filter`) que hace referencia a una función que ordena la lista de elementos obtenidos.
- Además el método permite especificar como requisito paramétrico una lista de elementos concretos (que debe ser compatible con la cadena anterior) y que es adecuada para que el usuario final indique exactamente qué elementos quiere transformar.
- `isystem_find_adaptation_actions_method`, utiliza las diferencias entre los valores de los atributos de las descripciones del caso y la consulta, y la representación explícita de cómo los elementos de la solución se ven afectados por estas diferencias. Esto se corresponde con dependencias (`E1 depends_on E2`) donde `E2` pertenece a la descripción y `E1` a la solución. El método incluye como requisito de conocimiento el uso de la relación `depends_on` de CBROnto, bien directamente, bien a través de la clasificación.

Estos métodos se relacionan con los mecanismos descritos en el Capítulo 2 para identificar qué partes de la solución deben ser modificadas. En particular, el método `isystem_find_adaptable_actions_method` se relaciona con la aproximación descrita en [Kolodner93] que se basa en las diferencias entre las descripciones entre la consulta y el caso recuperado junto con la representación explícita de conexiones entre los elementos de la solución y de la descripción que sean dependientes. Además, también es adecuado cuando la descripción y la solución comparten elementos, considerándose una forma de dependencia trivial. El otro mecanismo descrito en [Kolodner93] se basa en utilizar listas de comprobación proporcionadas por el usuario, lo que se corresponde con el método `ifix_adaptable_items_Method` donde la lista de comprobación la proporciona el diseñador o el método `iuser_find_adaptable_actions_Method` si es el usuario final el encargado de identificar los elementos susceptibles de adaptación.

Cada uno de los métodos anteriores generará una lista de acciones (instancias del concepto `Transformation_Action`), cada una de las cuales hace referencia al elemento que estará involucrado (atributo `item`), y la transformación que se llevará a cabo sobre él (atributo `transformation-spec` cuyo relleno es una instancia del concepto `Transformation`). Para determinar qué paso de transformación concreto se llevará a cabo sobre cada elemento se utilizan las pautas de aplicabilidad de las transformaciones involucradas en la estrategia de adaptación utilizada. Es decir, en primer lugar se identifica una lista de elementos a transformar y sobre cada uno se aplicará aquella transformación de la estrategia de adaptación cuyas características de aplicabilidad son satisfechas por el elemento en cuestión. Si el elemento no satisface ninguna de las características de aplicabilidad especificadas por la estrategia se aplicará el operador que se haya aplicado al individuo del que depende, o alguno de los operadores de la estrategia para los que no se incluyan una descripción de aplicabilidad.

El método `isystem_find_adaptable_actions_method` se puede configurar de forma que si varias transformaciones son aplicables sobre el mismo elemento, pueda seleccionar una de ellas aleatoriamente, o según el peso asociado a la transformación (que se puede fijar en cada transformación, en el atributo `weight`, o aprenderse en las distintas interacciones con el usuario), o para que sea el usuario final el encargado de elegirla (y pesar la transformación elegida). Para ello en el individuo que representa los requisitos de diseño del método se incluye un

atributo `select_transformation_by` cuyo relleno es una de las instancias del concepto `Select_Transformation`: `random`, `user` o `use-weight`.

4.1.3.3 Aplicación de las estrategias de adaptación

Cada estrategia de adaptación determina una lista de transformaciones a aplicar sobre un caso recuperado. Además, cada transformación puede hacer referencia a ciertas condiciones de aplicabilidad y a la estrategia de búsqueda de la información involucrada en la transformación. Como hemos visto, basándonos en la estrategia de adaptación, la subtarea `ifind_adaptation_actions_task` identifica una lista de acciones de transformación que se llevarán a cabo sobre la solución del caso, así como los elementos involucrados en estas transformaciones.

El método `imodify_solution_method` (asociado a la tarea `imodify_solution_task`) resuelve de forma cíclica la subtarea `apply_transformation_task` a través del método `apply_transformation_method`, encargado de aplicar una a una las transformaciones de la lista de acciones (transformación-elemento) que recibe como requisito de secuencia. Cada transformación es una instancia del concepto `Transformation` que además de la acción a realizar indica cómo llevarla a cabo, por ejemplo, cómo encontrar los sustitutos.

Observamos que la lista incluye únicamente a los operadores de sustitución y borrado, que involucran elementos de la solución del caso a adaptar, y no al de inserción que nunca involucrará a un elemento existente del caso. Por esta razón, el método `imodify_solution_method` después de procesar la lista de elementos, aplicará los operadores de inserción de elementos especificados en la estrategia de adaptación.

Para aplicar el operador de inserción, la estrategia de adaptación puede incluir una referencia al individuo concreto que se inserta, y el camino de relaciones que conectará el individuo caso con el nuevo individuo, o una referencia a los mecanismos de búsqueda que recuperan el elemento a añadir.

Para aplicar el operador de borrado no se necesita especificar conocimiento adicional ya que el resultado de aplicarlo es la supresión de un individuo en todas sus apariciones en el caso. Describimos a continuación el operador de sustitución.

Operador de sustitución de elementos

Para aplicar el operador de sustitución el método `apply_transformation_method` necesita cierto conocimiento adicional, por ejemplo, la especificación del lugar en el que llevar a cabo la búsqueda o cómo valorar si un sustituto es o no adecuado.

Del mismo modo que una estrategia de búsqueda para inserción de elementos, una estrategia de búsqueda para la sustitución de elementos puede usar una descripción en términos de las operaciones primitivas de acceso a la base de conocimiento que son necesarias para encontrar valores en la memoria, y del individuo a partir del cual se aplica la estrategia de búsqueda: el caso actual (`current-case`), el individuo consulta (`current-query`) o un individuo cualquiera del dominio.

Adicionalmente, una estrategia de búsqueda para la sustitución de elementos puede hacer referencia a alguno de los métodos de búsqueda (instancias de `Search_Substitutes_Method`) que son individuos que representan a los mismos métodos que describimos para la recuperación de casos, eliminando las subtareas innecesarias y ajustando los requisitos que pueden variar ligeramente al usarse los métodos en otro contexto. Este método será en cada paso el encargado de buscar el sustituto adecuado para un elemento que satisfaga los requisitos de aplicabilidad de la estrategia. Si se utiliza un método, el diseñador debe configurarlo adecuadamente igual que haría en la recuperación de casos. Se utilizan individuos distintos que pue-

den configurarse de forma independiente. Por ejemplo, para el método de recuperación por cómputo de similitud o por criterios de relevancia, el diseñador debe incluir las medidas de similitud numéricas o los criterios de relevancia adecuados para buscar en el modelo del dominio un sustituto para un cierto elemento.

Una estrategia de adaptación puede hacer referencia a varias estrategias de búsqueda que se pesan convenientemente. Para buscar sustitutos primero se usarán las estrategias en orden dado por los pesos (decreciente) hasta que alguna de ellas encuentre un sustituto.

Este tipo de estrategias de búsqueda están relacionadas con alguna de las que describimos en el Capítulo 2. Por ejemplo, el uso del método de búsqueda de sustitutos por reconocimiento de instancias configurado adecuadamente se corresponde por un lado con la *reinstanciación* empleada en el sistema CHEF [Hammond89] que sustituye un elemento por otro utilizando los papeles que juega el elemento, que se determinan en función de la clasificación en la jerarquía del elemento que quiero sustituir. Un elemento es sustituido por otro elemento "hermano", es decir, clasificado de la misma forma en la jerarquía. Además, también se corresponde con la *búsqueda local* que consiste en buscar en una jerarquía de abstracción un elemento cercano al que quiero sustituir. Diferentes sistemas de CBR han utilizado búsqueda local con diferentes guías para la navegación por la taxonomía o para indicar hasta dónde llegar, lo que se corresponde con la configuración del operador GTO (como se explicó en el Apartado 3.2).

La especificación de una estrategia de búsqueda para la sustitución de elementos en términos de las operaciones primitivas de acceso a la base de conocimiento se relaciona con la *búsqueda especializada*, utilizada por ejemplo en el sistema SWALE [Kass90], en la que se dan instrucciones sobre *cómo* buscar en una jerarquía de abstracción el elemento sustituto que se necesita usando heurísticas de búsqueda especializada que señalan las zonas de la jerarquía donde es probable que se pueda encontrar lo que se busca.

4.2 Método de adaptación basado en sustituciones

El método de adaptación basado en sustituciones –representado por el individuo canónico `iAdapt_by_Substitution_Method`– se basa en adaptar la solución del caso recuperado realizando sustituciones genéricas guiadas por el conocimiento del dominio. Descompone la tarea de modificación de la solución en las tres subtareas que hemos descrito para el método de adaptación basado en estrategias de adaptación específicas: buscar una estrategia de adaptación adecuada para la situación, utilizar la estrategia anterior para identificar una lista de acciones de transformación que se llevarán a cabo sobre la solución del caso y aplicarlas para resolver las discrepancias.

La única diferencia se refiere a la primera subtarea `Select_Strategy_Task` que se representa a través de la instancia `isubstitution_strategy_task` –ligada al método `isubstitution_strategy_method`– cuyo objetivo es recuperar una estrategia genérica de sustitución de elementos que está predefinida, en vez de una estrategia específica definida durante la fase de diseño del sistema como en el método anterior:

```
(tellm (:about Substitution-Strategy Adaptation_Strategy (transformation-spec subst)))
(tell (:about subst Transformation (operator substitute) (applicability 'Thing)
                                    (search-strategy gse)))
(tell (:about gse Method-Search-Strategy
               (search-method iSearch_Instance_Classification_Method)))
```

Esta estrategia de búsqueda genérica lleva a cabo un tipo de búsqueda local de sustitutos ya que hace referencia a un método de búsqueda que se corresponde con el método de recuperación de casos por reconocimiento de instancias. De esta forma, se buscarán sustitutos que estén clasificados *cerca* del individuo que vamos a sustituir. De todas formas, los requisitos de entrada, en particular los requisitos de diseño, del método `isubstitution_strategy_method` permiten modificar la estrategia de búsqueda genérica, en concreto las condiciones de aplicabilidad y el modo de buscar sustitutos, mediante cualquier instancia del concepto *`Search-Strategy`* tanto una instancia de *`Simple-Search-Strategy`* para definir la búsqueda de sustitutos en términos de operaciones primitivas de acceso a memoria, o de *`Method-Search-Strategy`* para usar un método de búsqueda.

Las dos subtareas adicionales derivadas de la aplicación del método de adaptación basado en sustituciones son comunes al método de adaptación basado en estrategias específicas, ya que, utilizando la estrategia concreta (en este caso la estrategia de sustitución genérica), la tarea `Select_Discrepancy_Task` (instancia `ifind_adaptation_actions_task`) identifica la lista de acciones de transformación que se llevarán a cabo sobre la solución del caso y los elementos involucrados en estas transformaciones, y la subtarea `Modify_Solution_Task` (a través de su instancia `imodify_solution_task`) se encarga de resolver la lista de discrepancias aplicando las transformaciones indicadas por la estrategia de adaptación (en este método serán únicamente sustituciones).

4.2.1 Requisitos del método

El método `iAdapt_by_Substitution_Method` no incluye requisitos de aplicabilidad adicionales, ya que la estrategia genérica de sustitución está predefinida en CBROnto. Como requisito de aplicabilidad comparte el individuo `iReuse_Application` con el método de adaptación de casos que representa la existencia de solución en los casos.

Respecto a los requisitos de entrada, no se especifican requisitos de conocimiento, ni de diseño, ni paramétricos. Los requisitos de secuencia se definen mediante el individuo `iAdapt_Solution_Sequence` que especifica la existencia del atributo `casetoAdapt`, cuyo relleno es un individuo (instancia de `Case_with_Solution`) que incluye la descripción de la consulta y la solución del caso recuperado, y el atributo caseList con la lista de casos recuperados.

5. Revisión

Una vez que la solución ha sido adaptada, la tarea de revisión se encarga de evaluar su adecuación a la situación actual. Dependiendo del tipo de adaptación, muchas aproximaciones consideran la revisión como una subtarea de la adaptación, en vez de cómo una tarea en sí misma. Aunque en nuestro esquema existe un proceso de revisión local que forma parte de la adaptación (subtarea `itransformation_local_revisión_task` que se encarga de validar o rechazar cada uno de los pasos de transformación de la solución), la tarea de revisión que ahora nos ocupa se refiere a una revisión global del resultado obtenido.

Una de las opciones ofrecidas en CBROnto, llevada a cabo en la mayoría de los sistemas, consiste en delegar la tarea de revisión en el usuario del sistema, que valida la solución generada por el mismo. Si además del método de revisión manual queremos incorporar métodos de revisión automática surge la necesidad de disponer de una representación explícita de la tarea que resuelve el sistema, por ejemplo, en términos de los objetivos que satisface un caso correcto. Dado un caso adaptado, también debemos ser capaces de valorar qué objetivos satisface y compararlos con los objetivos determinados para la corrección.

Respecto a la representación explícita de la tarea que resuelve un cierto sistema, existen trabajos que proporcionan un vocabulario genérico de descripción de tareas que es aplicable para un rango de distintos dominios de aplicación [Seta *et al.* 99]. Aunque nosotros no hemos utilizado esta aproximación, sí hemos incluido un método básico de revisión automática que se basa en representar la tarea[7] a resolver como una condición de corrección sobre los casos adaptados. La condición de corrección se define como una conjunción de conceptos del dominio, especificando a qué conceptos debe pertenecer un caso adaptado correcto. El método de revisión automática comprueba si el caso obtenido satisface o no los objetivos, es decir, si es una instancia de esos conceptos. Esto se lleva a cabo aprovechando el mecanismo de reconocimiento de instancias de LOOM. Además la representación explícita y declarativa de estos criterios permite explicar por qué el individuo no pertenece a un cierto concepto.

Se pueden observar numerosas similitudes entre este método de revisión y el método de adaptación basado en estrategias específicas del dominio. La idea subyacente es dar soporte a dos aproximaciones de diseño de sistemas CBR, una primera aproximación en la que se da más peso a la tarea de adaptación mientras que el proceso de revisión es simple o externo al sistema CBR, y otra aproximación en la que la tarea de adaptación se resuelve de forma simple y es la tarea de revisión la que se encarga de las comprobaciones más exhaustivas.

Con esta estructura de tareas en los procesos de adaptación y reparación se puede optar por varios tipos de sistemas CBR. Por un lado sistemas en los que las transformaciones específicas del dominio se lleven a cabo en la tarea de adaptación, es decir, la adaptación es compleja, utiliza conocimiento del dominio en forma de estrategias de adaptación y asegura que el resultado generado es correcto con respecto a los objetivos del sistema, es decir, nunca realiza una transformación que viole los objetivos de corrección. En este caso la evaluación (automática) siempre tiene éxito y la reparación automática deja de tener sentido. Sólo se incluiría un método de revisión manual que valida la solución de forma externa para valorar el nivel de satisfacción del usuario final. La otra opción sería la de diseñar sistemas CBR en la que la tarea de adaptación se resuelve de forma genérica, es decir, sin conocimiento específico del dominio. De esta forma, es el proceso de revisión automática el que mediante las estrategias de reparación específicas del dominio se encarga de generar soluciones correctas.

La idea que justifica los dos tipos de estrategias, es la posibilidad de descomponer las transformaciones sobre el caso recuperado en varias etapas, una primera etapa (adaptación) de modificaciones guiadas por la consulta, es decir, orientadas a satisfacer los requisitos pedidos y otra etapa (revisión) para reparar la solución obtenida, que aunque es correcta en cuanto a los requisitos de la consulta, puede no ser correcta respecto a los requisitos del propio dominio o de la aplicación. En el ejemplo del sistema de movimiento de bloques, las estrategias de adaptación indican cómo modificar el caso recuperado para que los bloques estén en los contenedores adecuados según se indica en la consulta. Sin embargo, las restricciones del dominio obligan a considerar la capacidad de cada contenedor, es decir, no se considerará válida una solución que debido a sus movimientos sobrepase la capacidad de un contenedor. Durante la tarea de revisión, se evalúa este posible fallo y, en caso de que ocurra, se aplica una estrategia de reparación.

El siguiente apartado describe los aspectos relativos a los requisitos del método y la resolución de las subtareas que genera. El Apartado 5.2 se encarga de los tipos de fallos y las estrategias de reparación. El Capítulo 6 incluye un ejemplo completo de uso de este método.

[7] No se corresponde con las tareas del CBR incluidas en CBROnto sino a una descripción de los objetivos del sistema en términos del dominio.

5.1 Resolución de las subtareas y requisitos

Como se describió en el Capítulo 4 existe un único método de revisión –representado por el individuo canónico `iRevise_Method`– que descompone la tarea de revisión en dos subtareas que se resuelven secuencialmente: revisión automática y revisión manual. Para cada una de las dos subtareas anteriores existen sendos métodos que las resuelven: método de revisión automática –que a su vez genera dos subtareas: evaluación y reparación– y método de revisión manual.

El método de evaluación automática –`iSystem_Evaluation_Method`– es un método de resolución que se basa en clasificar el caso adaptado y comparar los conceptos bajo los que está clasificado con la clasificación del caso recuperado. En función de las diferencias en la clasificación se identifican los problemas o fallos que tiene el caso propuesto. Una vez identificados los problemas comienza la resolución de la subtarea de reparación automática –`iSystem_Repair_Method`– en la que los problemas encontrados sirven de índices para recuperar las estrategias de reparación adecuadas. Realmente el flujo de control define un ciclo entre las tareas de evaluación y reparación, de forma que en cada vuelta se identifica un fallo y se intenta su reparación.

Al igual que el método de adaptación utilizando estrategias, el método de reparación automática descompone la tarea de reparación en dos subtareas: encontrar estrategia y aplicar estrategia. El método que aplica la estrategia devuelve un resultado de éxito o fallo, de forma que si la estrategia falla y no es capaz de reparar el error, dejaremos que lo arregle el usuario en la tarea de revisión manual. Aunque este método se define de forma genérica e independiente del dominio, depende claramente del conocimiento específico del que disponga. Es decir, el método funcionará en base a los tipos de problemas y las estrategias de reparación que se deben definir explícitamente para cada dominio concreto. Además, el método requiere una representación explícita de la tarea que resuelve el sistema en términos de los objetivos que satisfará un caso correcto. En nuestro esquema gracias a la representación explícita de conocimiento, esto no supone incluir conocimiento adicional y que los objetivos que debe satisfacer un caso adaptado se definen en términos de clasificación en el modelo del dominio, en concreto, los objetivos son los conceptos a los que pertenece el caso recuperado.

De esta forma, dado un caso adaptado podemos valorar qué objetivos satisface (a qué conceptos pertenece) y compararlos con los objetivos determinados para la corrección (conceptos de los que el caso recuperado es instancia). El método genérico utiliza el caso y la descripción de la corrección para revisarlo.

Los requisitos de aplicabilidad del método de revisión automática coinciden con sus requisitos de conocimiento y suponen la existencia previa de los tipos de fallo (instancias de `Fail_Type`) y de las estrategias de reparación (instancias de `Repair_Strategy`) en las que se basa el proceso de revisión automática y que se describen en el siguiente apartado.

Respecto a los requisitos de entrada, no se especifican requisitos paramétricos, ni de diseño y los requisitos de secuencia especifican la existencia del atributo `adaptedCase`, cuyo relleno es un individuo (instancia de `Case_with_Solution`) que representa el caso adaptado que hay que revisar.

El método de revisión genera como salida el individuo caso revisado –atributo `revised-Case`– y un indicador de si el caso satisface o no los requisitos de corrección establecidos –atributo `revise-result` (`success` o `failure`).

5.2 Tipos de problemas y estrategias de reparación

De forma similar a los tipos de problemas de adaptación, los tipos de problemas de reparación (o fallos) son específicos del dominio de cada aplicación concreta. El diseñador identificará los problemas a resolver basándonos en la clasificación del caso adaptado. Por ejemplo, en un sistema CBR para una agencia de viajes, el diseñador puede definir que ocurre un problema cuando, debido a las transformaciones llevadas a cabo durante la adaptación en las que podemos haber cambiado el `destino` o la `duración`, un individuo `Viaje` se clasifique como instancia del concepto del dominio `Viaje_Caro` o `Viaje_Muy_Caro`, y deje de pertenecer a `Viaje_Barato`.

Con esta información, COLIBRI representará cada tipo de problema como un concepto (subconcepto del concepto FAIL_TYPE de CBROnto) con una definición adecuada para que el individuo involucrado en el problema se reconozca como una instancia de ese concepto cuando ocurre dicho problema. En el ejemplo, se define un concepto `Problema_Caro` y, utilizando la información dada por el diseñador, el sistema clasificará en él a los individuos viaje que se reconozcan como `Viaje_Caro` o `Viaje_Muy_Caro`.

Cada tipo de problema es anotado con una estrategia de reparación para ese problema específico. Las estrategias de reparación son instancias del concepto de CBROnto `repair_strategy` y se unen con un tipo de fallo a través de la relación de CBROnto `has_repair_strategy`.

El método que encuentra la estrategia buscará las instancias de la jerarquía de tipos de fallo y recupera la estrategia del concepto más específico que tenga definida una estrategia de reparación. En el siguiente paso, el método (`apply-strategy`) aplica la estrategia recuperada al individuo clasificado bajo el tipo de problema.

El método `apply-strategy` recupera y aplica la función Lisp y/o el método asociados a la estrategia. Si la reparación falla el individuo involucrado, que estará clasificado bajo un concepto que representa el tipo de fallo, se marca como fallido (usando la relación `fail-repair`) y será el usuario el que se encargue de la reparación en la subtarea de reparación manual. Si la reparación tiene éxito el individuo involucrado deja de estar clasificado bajo el concepto tipo de fallo.

En el ejemplo de los viajes, el diseñador puede definir una estrategia de reparación que se asociará al concepto `Problema_Caro`, creando y configurando un método (instancia del concepto `find_adaptable_parts`) para que sustituya el destino, y el criterio de sustitución será encontrar un destino más cercano.

6. Aprendizaje

Cuando la validación del caso que resulta de la tarea de reparación es positiva se resuelve la tarea de aprendizaje de la experiencia. Como se describió en el Capítulo 4, CBROnto define un método que resuelve la tarea de aprendizaje descomponiéndola en dos subtareas que se resuelven de forma secuencial: aprender un nuevo caso que se incorpora a la base de casos y aprender conocimiento de recuperación, adaptación y revisión de casos, añadiendo resultados de éxito o fallo en el caso recuperado y pesando positiva o negativamente las estrategias de adaptación y reparación utilizadas.

6.1 Resolución de las subtareas y requisitos

El método `iRetain_Method` no incluye requisitos directamente, aunque cada uno de los métodos que resuelven las subtareas de aprendizaje recibe como requisitos de secuencia la información de los procesos de recuperación, adaptación y revisión que sea susceptible de ser aprendida y que se propaga a través de las salidas de los métodos. Una característica común es que para que se lleve a cabo el aprendizaje es necesario que exista un proceso de revisión positiva, bien manual o bien automática.

La primera subtarea se encarga de aprender un nuevo caso, es decir, incorporarlo a la base de casos. De esta forma el sistema CBR actualiza su conocimiento en base a nuevas experiencias que permitirán abordar futuros problemas. Este aprendizaje sólo se produce si la validación del caso que resulta de la tarea de reparación es positiva. Es decir, que el método –`iretain_case_method`– que resuelve esta subtarea tiene como requisitos de secuencia la existencia de los atributos `revise-result` y `revisedCase` que es el caso final que se ha obtenido como resultado. Recordamos que el atributo `revise-result` indica si el caso satisface los requisitos de corrección establecidos. Sólo si este atributo tiene el valor `success` el caso es *aprendido*. Un caso aprendido pasa a formar parte de una base de casos temporal, es decir, no se añade directamente a la base de casos que maneja el sistema CBR.

Para que los casos de la base de casos temporal pasen a la base de casos definitiva, que se usa en la resolución de problemas, se debe resolver la tarea de mantenimiento de la base de casos (`CB_Maintenance_Task`). Como veremos en el Capítulo 6 los usuarios finales de la aplicación sólo pueden invocar la resolución de la tarea `CBR_Task`, es decir, la tarea de resolución de problemas con la que se empieza un ciclo CBR. La resolución del resto de las tareas –como la adquisición de casos, la integración de conocimiento o el mantenimiento– que se consideran *tareas de diseño*, se reserva a usuarios del tipo diseñador (que está predefinido en CBROnto).

La segunda subtarea se encarga de aprender conocimiento de recuperación, de adaptación y de revisión de casos. Para ello debe recibir de esas tareas el conocimiento que se incorporará al conocimiento del dominio y de la aplicación con el que razonan los métodos. En concreto, el método –`iretain_knowledge_method`– que resuelve esta subtarea deriva a su vez tres subtareas: `iretain_retrieval_knowlege_task`, `iretain_reuse_knowledge_task` e `iretain_revise_knowledge_task`, cuya resolución no es obligatoria pero que se pueden resolver por los siguientes métodos, respectivamente:

- `iretain_retrieval_knowlege_method` permite aprender un resultado de éxito o de fallo en el caso recuperado. Para ello recibe como requisito de secuencia una referencia al caso recuperado, en el atributo `retrieved_case`, y el atributo `revise-result` cuyo valor determina el resultado que se aprende (éxito o fallo) junto con una referencia a la consulta que causó la recuperación de este caso y que también se añade como parte del resultado (la consulta forma parte del contexto).
- `iretain_reuse_knowledge_method` permite aprender pesos positivos o negativos sobre las estrategias utilizadas durante la resolución de la tarea de adaptación. Para ello recibe como requisito de secuencia una referencia a las estrategias utilizadas –atributo `adaptation_strategy`– y el atributo `revise-result` cuyo valor determina si el aprendizaje es positivo (sumamos 1 al peso de la estrategia) o negativo (resta 1).
- `iretain_revise_knowledge_method` es análogo al anterior pero permite aprender pesos asociados a las estrategias utilizadas durante la tarea de revisión (si existen). Para ello recibe como requisito de secuencia la referencia a las estrategias utilizadas

–atributo `repair_strategy`– y el atributo `revise-result` cuyo valor determina el tipo de aprendizaje (positivo o negativo).

7. Resumen y conclusiones del capítulo

En este capítulo hemos descrito los métodos que se incluyen en la biblioteca de PSMs de CBROnto. En concreto hemos descrito cada uno de los métodos que actualmente están formalizados en la biblioteca y cómo se organizan en torno a las tareas CBR que resuelven. Además hemos descrito algunos aspectos relacionados con su representación en LOOM. El lector interesado puede complementar el contenido de este capítulo a través de los Apéndices B y C. El Apéndice B incluye la formalización de CBROnto en LOOM. El Apéndice C incluye una descripción textual de todos los métodos de la biblioteca de PSMs.

Los métodos que hemos incluido en la versión actual de CBROnto son adecuados para los sistemas KI-CBR ya que sacan partido del conocimiento general sobre el dominio del que disponen los sistemas diseñados. Otro punto en común entre los métodos de la biblioteca es que se definen en base a los mecanismos de razonamiento del sistema de DLs en el que se basa nuestra propuesta. La biblioteca de métodos permite añadir nuevos métodos que resuelven una cierta tarea manteniendo el mismo esquema general de resolución y sin afectar al resto de los métodos.

Aunque hemos intentado proporcionar ejemplos para cada uno, el objetivo de este capítulo es la descripción del comportamiento de los métodos desde un punto de vista genérico, independientemente de su uso en una aplicación concreta. En el siguiente capítulo veremos el proceso de diseño de aplicaciones y cómo utilizar la biblioteca de métodos para diseñar una aplicación con COLIBRI.

Capítulo 6

DESARROLLO DE APLICACIONES CBR USANDO COLIBRI/CBRONTO

1. Introducción

Como hemos descrito en los capítulos previos de esta memoria, en esta tesis hemos investigado cómo sintetizar aplicaciones KI-CBR combinando componentes reutilizables independientes entre sí. El sistema COLIBRI propone un modelo de desarrollo de aplicaciones KI-CBR que se basa en reutilizar, por un lado, el conocimiento acerca del CBR en sí mismo, a través de la terminología, tareas y métodos CBR de CBROnto; y, por otro lado, ontologías con conocimiento terminológico del dominio de una biblioteca de ontologías [Díaz&González00a] [Díaz&González01d] [Díaz&González01e] [Díaz&González01f] [Díaz&González01g] [Díaz&González02].

Después de describir la terminología y la biblioteca de métodos de CBROnto -en los Capítulos 4 y 5- en este capítulo explicamos cómo utilizar CBROnto en el contexto de COLIBRI. La idea subyacente a nuestra aproximación es que el trabajo invertido al crear un sistema CBR puede ser reutilizado. En general, esta aproximación se usa, de forma más o menos explícita, en cualquier desarrollo de un sistema CBR. Un ejemplo de reutilización implícita se produce cuando un diseñador estudia la literatura existente para utilizar alguna de las técnicas clásicas en el área del CBR que puede ser adecuada para la aplicación que quiere desarrollar. Nuestra aproximación, y en general la que se usa en cualquier entorno de desarrollo de aplicaciones CBR, propone un tipo de reutilización más explícita, a través de implementaciones previas de módulos que serán invocados por el nuevo sistema en tiempo de ejecución.

El Apartado 2 describe la arquitectura del sistema COLIBRI. El Apartado 3 describe el proceso general de diseño de aplicaciones con COLIBRI que ejemplificamos en el Apartado 4 con el desarrollo de una aplicación de generación de poesías en castellano [Díaz *et al.* 02].

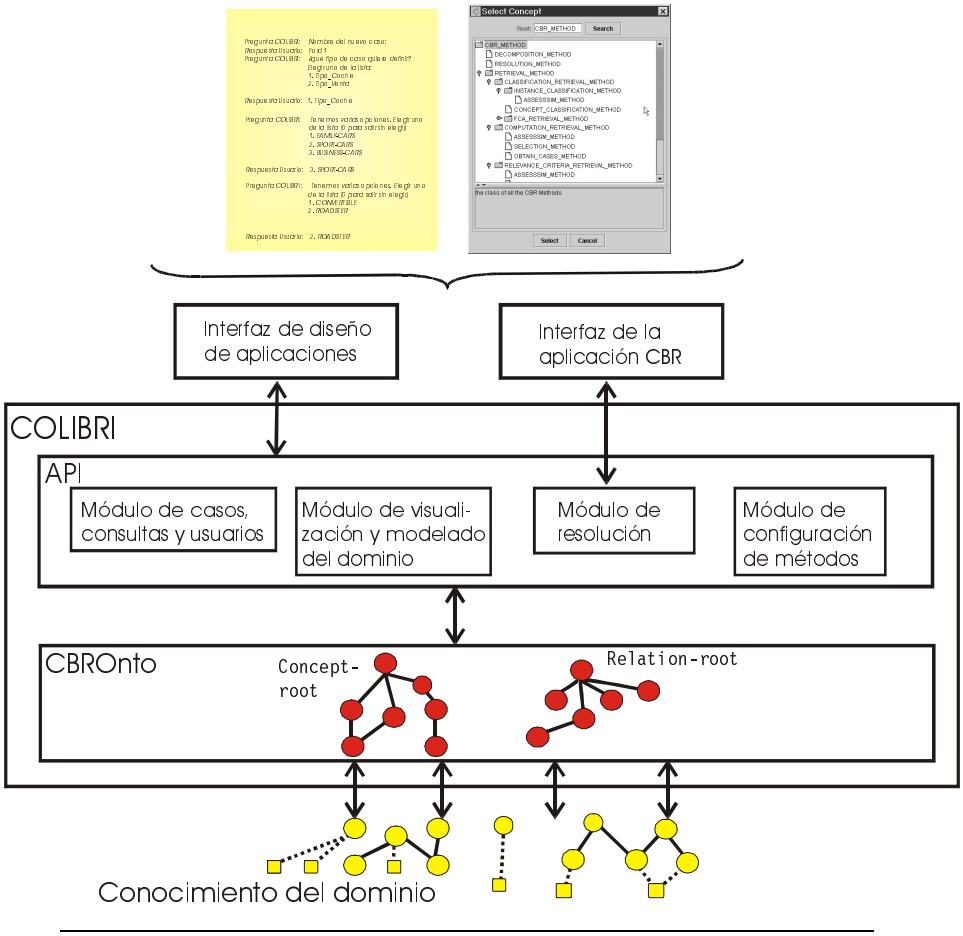


***Figura 6-1**. Arquitectura de COLIBRI*

2. Arquitectura del sistema COLIBRI

COLIBRI está implementado en LOOM y Lisp (el lenguaje de programación de propósito general asociado a LOOM). En la arquitectura del sistema (Figura 6-1) se distingue:

- Un módulo que contiene la formalización de CBROnto, la ontología de CBR en la que se basa COLIBRI.
- La API de COLIBRI dividida en varios módulos funcionales que se encargan de la gestión del contenido de la base de conocimiento subyacente a la aplicación (compuesta por CBROnto más el modelo del dominio). Dotar a COLIBRI de una API facilita su uso, bien como un sistema independiente con interfaz o bien como un módulo que forme parte de otras aplicaciones. Cada uno de los módulos siguientes agrupa ciertas funciones de la API de COLIBRI:
  - Modulo de visualización y modelado del dominio: agrupa las funciones relacionadas con la adquisición y visualización del conocimiento, y la modificación de las jerarquías de términos para la integración de conocimiento.

  - Modulo de definición de casos, consultas y usuarios: agrupa las funciones relacionadas con la definición de las estructuras o tipos de casos, tipos de consultas y tipos de usuarios de la aplicación.
  - Módulo de configuración de métodos: agrupa las funciones necesarias para configurar las tareas y los métodos a través de la especificación de sus requisitos.
  - Módulo de resolución de problemas: agrupa las funciones necesarias para resolver tareas accediendo a los métodos configurados para ello. Los usuarios finales de una aplicación diseñada no tendrán acceso a la API completa sino únicamente a las funciones de este módulo de resolución.
- Dos módulos de interfaz, uno para generar la interfaz de diseño de aplicaciones y otro para generar la interfaz del usuario final de la aplicación. Las funciones de los módulos de interfaz invocan a las funciones de la API de COLIBRI para construir dinámicamente los diálogos de la interfaz utilizando el conocimiento subyacente. En la versión actual COLIBRI dispone de una interfaz textual para diseñar aplicaciones, aunque un prototipo de interfaz gráfica está siendo implementado en Java con mecanismos de intercambio de información en CORBA.

En el Apéndice D, se incluye el listado de funciones de la API de COLIBRI organizado según los módulos anteriores. El Apéndice B incluye la formalización en LOOM del módulo de CBROnto. Para cada aplicación diseñada, COLIBRI carga una copia nueva de la ontología completa a la que se añade el conocimiento del dominio integrado convenientemente.

3. Diseño de aplicaciones con COLIBRI

En este apartado describimos las fases del diseño de una aplicación CBR usando COLIBRI. Utilizamos un modelo de diseño en espiral en el que, de manera progresiva, se refinan los prototipos construidos (Figura 6-2). La ventaja de este modelo es que proporciona una forma rápida de prototipar aplicaciones en las que se pueden hacer pruebas (casi) desde el principio y estudiar la adecuación de los distintos métodos y configuraciones.

En el Capítulo 4 hemos descrito que, además de la jerarquía de tareas involucradas en la resolución de problemas (subconceptos de `CBR_TASK`), CBROnto también define otras tareas de primer nivel cuya resolución (igual que la de la tarea `CBR_TASK` para resolver problemas) se invoca externamente con una llamada al resolutor de tareas. Ejemplos de estas tareas son la adquisición de casos (`Case_Acquisition_Task`), la adquisición del modelo del dominio (`Domain_Model_Adq_Task`), la integración del conocimiento del dominio con el conocimiento de CBROnto (`Mapping_Task`), la organización de la base de casos en una estructura que facilite su acceso (`CB_Organization_Task`) y el mantenimiento de la base de casos (`CB_Maintenance_Task`). Salvo los métodos que computan el retículo AFC, para la tarea de organización de la base de casos, el resto de las tareas están ligadas a un mismo método genérico (`user_resolution_method`) que informa al diseñador de que él es el encargado de resolver la tarea externamente, y espera la respuesta de confirmación de que la tarea ha sido resuelta. La existencia de estas tareas tiene sentido en cuanto a la representación explícita de las tareas de diseño que se deben resolver. Estas tareas están relacionadas con las fases que un diseñador debe llevar a cabo para diseñar una aplicación KI-CBR utilizando COLIBRI y que describimos en los apartados siguientes:

- Adquisición del conocimiento del dominio (`Domain_Model_Adq_Task`). El objetivo es definir el conocimiento terminológico del dominio con el que razonará el sistema. Aunque idealmente podremos reutilizar conocimiento de una biblioteca de ontologí-

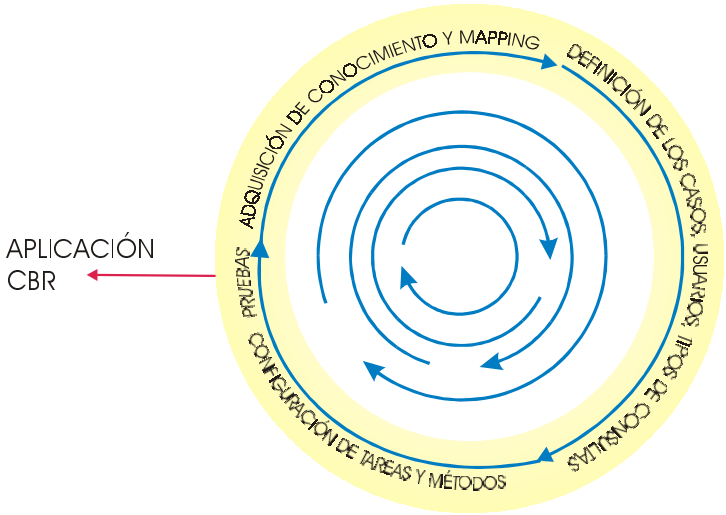


Figura 6-2. *Fases de diseño de una aplicación CBR con COLIBRI*

as (externa a COLIBRI), también está la opción de modelar desde cero una base de conocimiento *ad hoc* para el sistema. Para posibilitar el razonamiento de los métodos, este conocimiento se integrará (`Mapping_Task`) con el conocimiento de CBROnto.

- *Definición de los casos* (`Case_Acquisition_Task`). El objetivo de esta fase es definir la estructura de los casos usando (o no) los tipos predefinidos de CBROnto, y los casos concretos de estos tipos. La base de casos puede contener casos de distintos tipos, por ejemplo, con distinto nivel de granularidad.
- *Configuración de tareas y métodos CBR*. El objetivo de esta fase es seleccionar y configurar las tareas que se resolverán en el nuevo sistema, y reutilizar los métodos de la biblioteca de CBROnto.
- Durante la fase de *pruebas, depuración y modificaciones* el diseñador puede probar distintas configuraciones hasta obtener el comportamiento deseado. Además el diseñador puede definir características adicionales como tipos de consultas y tipos de usuarios y asociar a ellos distintas configuraciones de métodos. Por último, durante la fase de *mantenimiento* del sistema que el diseñador realizará periódicamente se decide la inclusión definitiva de los casos aprendidos en la tarea de aprendizaje de casos y se actualiza el conocimiento para evitar el degradamiento del rendimiento.

En los apartados siguientes describimos estas fases de diseño y las ejemplificamos con una aplicación CBR simple.

3.1 Adquisición del conocimiento del dominio

Esta fase del proceso de diseño de aplicaciones es necesaria en un entorno como COLIBRI cuyo objetivo es el diseño de aplicaciones KI-CBR que saquen partido del conocimiento del dominio. Por tanto, el propósito de esta fase es la adquisición y modelado del conocimiento del dominio del que dispondrá la aplicación CBR, además del de los casos, para razonar.

Como hemos descrito en los capítulos anteriores, nuestro modelo considera que el conocimiento ontológico disponible en bibliotecas públicas de ontologías es adecuado para diseñar sistemas KI-CBR. Además, el tipo de conocimiento terminológico presente en las onto-

logías del dominio es especialmente adecuado para los métodos de CBROnto, que se basan en los mecanismos de razonamiento de las DLs.

El objetivo de la fase de adquisición de conocimiento es obtener una representación explícita, expresada en LOOM, del conocimiento acerca del dominio de la aplicación CBR que vamos a diseñar. Las metodologías clásicas de Ingeniería del Conocimiento hacen una distinción explícita entre las ontologías del dominio y el conocimiento del dominio [Van Heijst *et al.* 97]. El conocimiento del dominio describe situaciones reales en un cierto dominio, mientras que una ontología del dominio restringe la estructura y los contenidos del conocimiento del dominio. Recordamos la definición de ontología propuesta en [Swartout *et al.* 97]*: "Una ontología es un conjunto de términos jerárquicamente estructurados para describir un dominio, que pueden ser utilizados como esqueleto en el que se fundamenta una base de conocimiento"* . En COLIBRI el conocimiento o modelo del dominio se formula usando el vocabulario de una ontología y satisface sus restricciones, aunque puede incluir nuevas instancias y conocimiento que describe el estado actual del dominio de aplicación. Por ejemplo, en una ontología de relaciones familiares encontraremos ciertos conceptos y relaciones como `personas`, `padres`, `hijos`, `hermano` o `descendiente`. Un modelo del dominio incluirá descripciones de familias concretas con individuos concretos, utilizando los conceptos y relaciones de la ontología y cumpliendo sus restricciones, por ejemplo, nadie es descendiente de sí mismo ni descendiente de sus descendientes, o si *x* es padre de *y* entonces *y* es hijo de *x*.

Proponemos el uso de bibliotecas de ontologías y entornos gráficos de edición y manipulación de las mismas que son externos a nuestro sistema. En particular nosotros hemos utilizado el editor y la biblioteca de ontologías del *Ontology Server* (OS) [Farquhar *et al.* 95] [Farquhar *et al.* 97] –y sus traductores a LOOM–, el editor WebODE [Arpirez *et al.* 01], y el entorno de integración de ontologías Chimaera [McGuiness *et al.* 00a y b], que hemos descrito en el Capítulo 3. COLIBRI requiere que el resultado sea guardado en archivos LOOM que COLIBRI incorpora a la base de conocimiento actual (que contiene una copia de CBROnto). Lamentablemente si ninguna de las ontologías disponibles es adecuada para la aplicación será necesario construir una base de conocimiento partiendo desde cero.

A efectos de su uso posterior, debemos tener en cuenta que el conocimiento del dominio se ha adquirido de forma independiente de los métodos que lo manejarán. Inicialmente podremos hacer distinción entre dos bases de conocimiento que se integrarán en una única: por un lado la que contiene el conocimiento del dominio y, por otro lado CBROnto, que contiene el conocimiento sobre CBR. Suponiendo que el modelo del dominio adquirido por reutilización de ontologías ofrece conocimiento suficiente para un cierto método, bien porque existían una o varias ontologías adecuadas que se han integrado, bien porque el conocimiento se ha refinado o construido *ad hoc* para la aplicación, la terminología de CBROnto sirve como puente para integrar el conocimiento del dominio con los métodos CBR que lo utilizan.

3.1.1 Integración del conocimiento del dominio con CBROnto

En el Capítulo 3 hemos descrito que una aproximación como la nuestra, que basa el diseño de aplicaciones en la reutilización de componentes, requiere conectar de algún modo las estructuras que existen en el modelo de conocimiento del dominio y las estructuras que "espera" el PSM. En nuestra aproximación llevamos a cabo esta conexión o integración de conocimiento a través de la relación de subsunción entre los términos del dominio y los términos de CBROnto [Díaz&González01e] [Díaz&González01f] [Díaz&González01g].

La integración del conocimiento del dominio con CBROnto estará guiada, en general, por los requisitos de los métodos. Por ejemplo, el requisito `CaseBase_With_Solution` del método

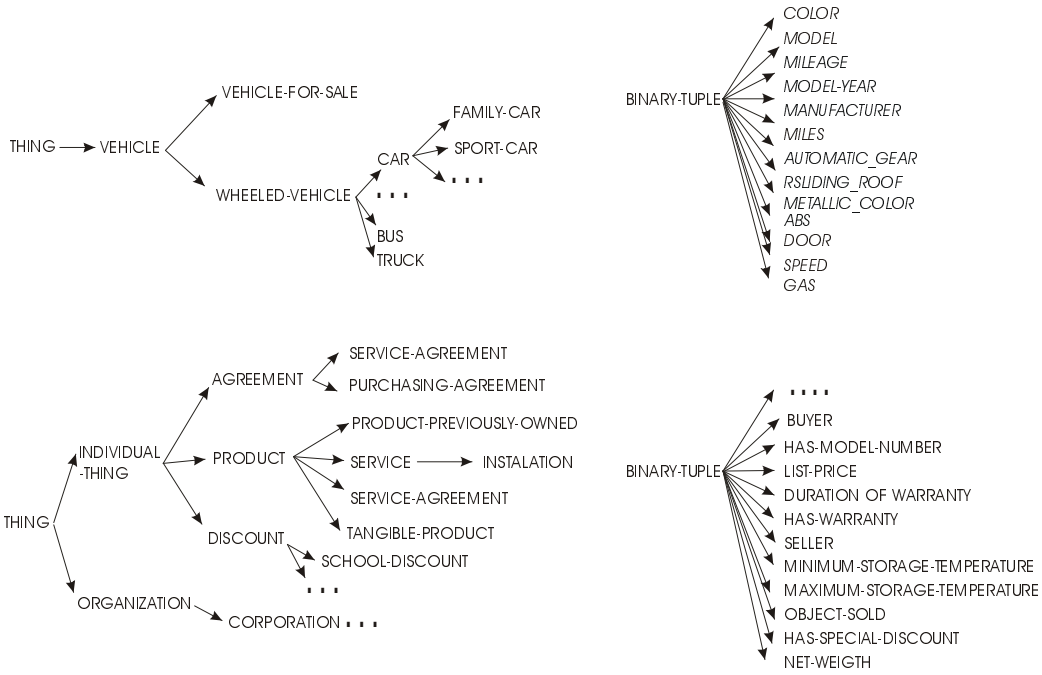


Figura 6-3. *Ontologías reutilizables en la aplicación*

de adaptación (ver Apéndice C) requiere que los casos tengan la relación (atributo) `has-solution`. Si ocurre que los casos tienen otra estructura, la clasificación de relaciones permite definir el puente para clasificar el atributo que relaciona un caso con su solución, bajo la relación `has-solution` de CBROnto, para que el método CBR sea capaz de razonar con los casos. En general, la fase de integración del conocimiento consiste en expresar los roles concretos que se identifican en un dominio con los roles genéricos identificados en CBROnto como caso, descripción, solución, resultado, actor, acción, objetivo, propiedad descriptiva, estructural, causal, etc.

Gracias a los mecanismos de herencia y propagación de restricciones de las DLs, el diseñador de la aplicación sólo debe clasificar manualmente los términos del nivel superior de las jerarquías del dominio. En esta fase del proceso de diseño de la aplicación no se pretende realizar una integración total, sino que se hará progresivamente y de forma guiada por el sistema. Normalmente el diseñador llevará a cabo parte de la integración durante esta fase de modelado del dominio donde se pueden identificar, por ejemplo, las dependencias entre los elementos del dominio (relaciones `depends_on`), pero también en otras fases del proceso de diseño, como la descripción de los casos en la que se utiliza el lenguaje de descripción de casos de CBROnto, y durante la selección y configuración de métodos, donde los requisitos de los métodos guiarán la definición de estas relaciones de integración.

3.1.2 Ejemplo de reutilización e integración de ontologías

Supongamos que queremos diseñar con COLIBRI una aplicación de compra/venta de coches. Para modelar el dominio tendremos en cuenta la disponibilidad de ontologías previas con conocimiento reutilizable y seleccionaremos las que sean potencialmente útiles. Por ejemplo, en el OS encontramos la ontología VEHICLE-ONTOLOGY con conocimiento general

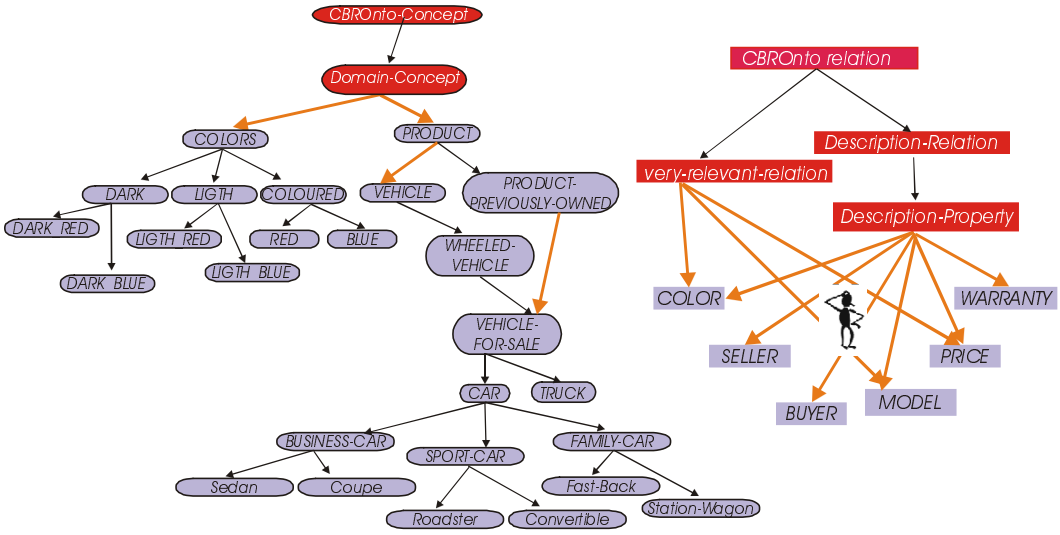


Figura 6-4. *Integración de conocimiento*

sobre vehículos y la ontología PRODUCT-ONTOLOGY con conocimiento sobre compra y venta de productos, que pueden ser adecuadas para nuestra aplicación (Figura 6-3).

Debido a que las ontologías, por definición, son generales y reutilizables, algunas veces no todas las definiciones serán útiles para el modelo del dominio que requiere la aplicación. Por ello el modelo resultante puede ser revisado para eliminar las definiciones innecesarias. Aunque este paso no es imprescindible, sí repercute en la eficiencia y calidad de los razonamientos llevados a cabo por la aplicación construida, porque el espacio de búsqueda será menor y contendrá únicamente términos relevantes. Por ejemplo, de la ontología de productos eliminaremos los conceptos `School-Discount` y `Service-Agreement`, y las relaciones `minimum-storage-temperature` y `maximum-storage-temperature`, que no tienen sentido en esta aplicación. De manera análoga, si el diseñador identifica términos que no están incluidos en la ontología deberá añadirlos. La ventaja clara de este esquema es que sólo se incluye parte del conocimiento, en concreto sólo las definiciones que la ontología no proporcione.

Para su uso en COLIBRI estas dos ontologías han sido integradas entre sí por clasificación para establecer, por ejemplo, que el concepto `Vehicle` es un subconcepto de `Product`. Como se muestra en la Figura 6-4 hemos añadido también una jerarquía taxonómica sobre colores, que integramos con la ontología existente estableciendo el concepto COLORS como el rango de la relación `color` de la ontología de vehículos (sustituyendo el rango *String* original). Aunque no se han incluido en el ejemplo, en el OS existen otras ontologías que también podrían incorporarse al conocimiento existente, por ejemplo, `Scalar-Quantities` contiene conocimiento sobre cantidades, incluyendo monedas y precios.

Después de esta fase, se dispone de un modelo de conocimiento inicial que captura las ideas esenciales necesarias para la aplicación. Con este conocimiento inicial podemos plantearnos algún tipo de integración inicial con CBROnto. Por ejemplo, la Figura 6-4 muestra un primer nivel de integración en el que los conceptos `COLORS` y `PRODUCT` se han clasificado como `Domain-Concept`, y las relaciones que describen a los coches se han clasificado bajo la relación `description-property` de CBROnto y las relaciones `color`, `model` y `price` se clasifican como relaciones muy relevantes del dominio (bajo `very-relevant-relation`).

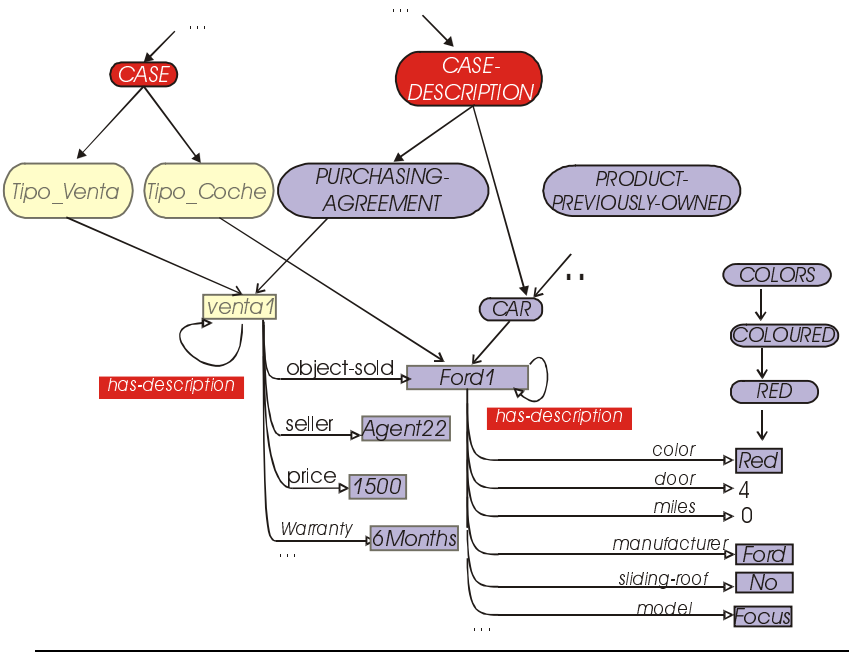


Figura 6-5. *Esquema de representación de casos*

3.2 Definición de los casos

Además del conocimiento general del dominio que utilizamos para enriquecer la aplicación, no debemos olvidar que el núcleo de conocimiento de un sistema CBR es su biblioteca de casos. COLIBRI y CBROnto permiten la definición de bases de casos heterogéneas y casos con estructuras complejas e incompletas. Como hemos visto en los Capítulos 4 y 5, los casos de la base de casos no tienen que compartir la misma estructura. Aunque no es obligatorio, el diseñador puede definir tipos de casos para agrupar aquellos casos que comparten una estructura común y puede aprovechar o no, a través de la clasificación, los tipos de casos predefinidos de CBROnto que hemos descrito en el Capítulo 4.

Se imponen dos únicos requisitos sobre la estructura de los casos: cada caso debe ser instancia del concepto `CASE` de CBROnto y tener un atributo `has-description` (ambos requisitos se pueden satisfacer directamente o a través de la jerarquía de subsunción). Manteniendo esto, se pueden diseñar distintos tipos de sistemas CBR, tanto sistemas en los que los casos no se agrupen por tipos y tengan estructuras totalmente heterogéneas, como sistemas en los que todos los casos comparten exactamente la misma estructura.

Una vez que el diseñador ha decidido la estructura de los casos, y ha definido los tipos de casos, se añade el conjunto inicial de casos concretos. Los casos se definen utilizando la terminología disponible sobre el dominio y la terminología de CBROnto (determinada por la estructura de los tipos de casos).

Durante este proceso COLIBRI guía la creación de nuevos casos a través de interfaces que se construyen de forma dinámica haciendo consultas genéricas a la base de conocimiento [Díaz&González00a]. Las preguntas y las opciones sobre las que selecciona el diseñador se obtienen mediante consultas a la base de conocimiento utilizando los términos de CBROnto,

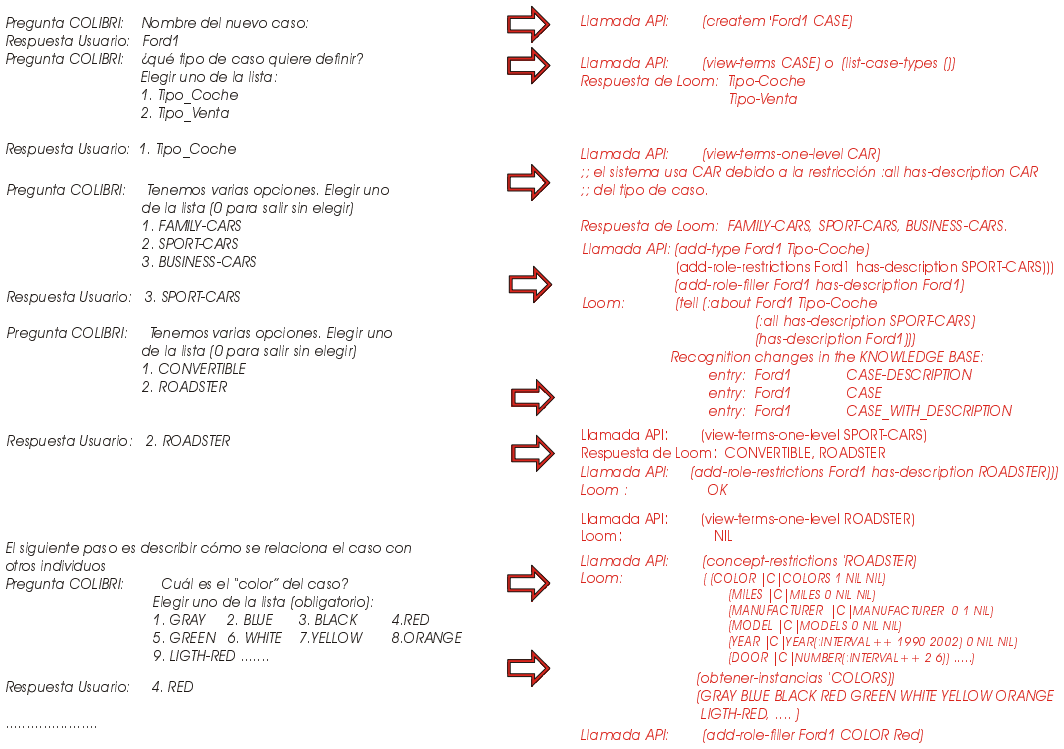


Figura 6-6. *Ejemplo de interfaz textual de definición de casos*

por tanto, el diálogo depende de la clasificación de los términos del dominio bajo los términos de CBROnto. El siguiente apartado muestra un ejemplo de este proceso.

3.2.1 Ejemplo de definición de casos

Supongamos que, como diseñadores del sistema CBR de venta de coches, hemos decidido crear dos tipos de casos. El primer tipo representa casos que son descripciones de coches que se usarán para hacer búsquedas de coches concretos que existan o coches que han estado aunque ya se hayan vendido. El segundo tipo representa casos que son operaciones completas de compra/venta de coches, incluyendo el vendedor, el comprador, el precio y las condiciones de la operación y el producto (es decir, el coche) involucrado en la venta. Observamos en el esquema de la Figura 6-5 que una parte del segundo tipo de casos son los coches, que son los casos del primer tipo. En ambos se decide una estructura simple de casos con descripción pero sin solución ni resultado.

El diálogo de la Figura 6-6 simula la interacción del usuario para definir un nuevo caso, teniendo en cuenta que las preguntas del sistema se generan dinámicamente haciendo consultas a la base de conocimiento a través de llamadas a funciones de la API de COLIBRI (parte derecha de la Figura 6-6) en la que los términos del dominio están clasificados bajo los términos adecuados de CBROnto.

En el diálogo el usuario elige que quiere crear un caso del tipo `Tipo_Coche` llamado `Ford1`. En respuesta a las preguntas del sistema, cuyas opciones coinciden con el recorrido de la jerarquía de subconceptos del concepto `CAR` (que es el concepto que define el tipo de descripción del caso) nivel por nivel, el usuario elige que el coche es de tipo `Roadster`, con lo

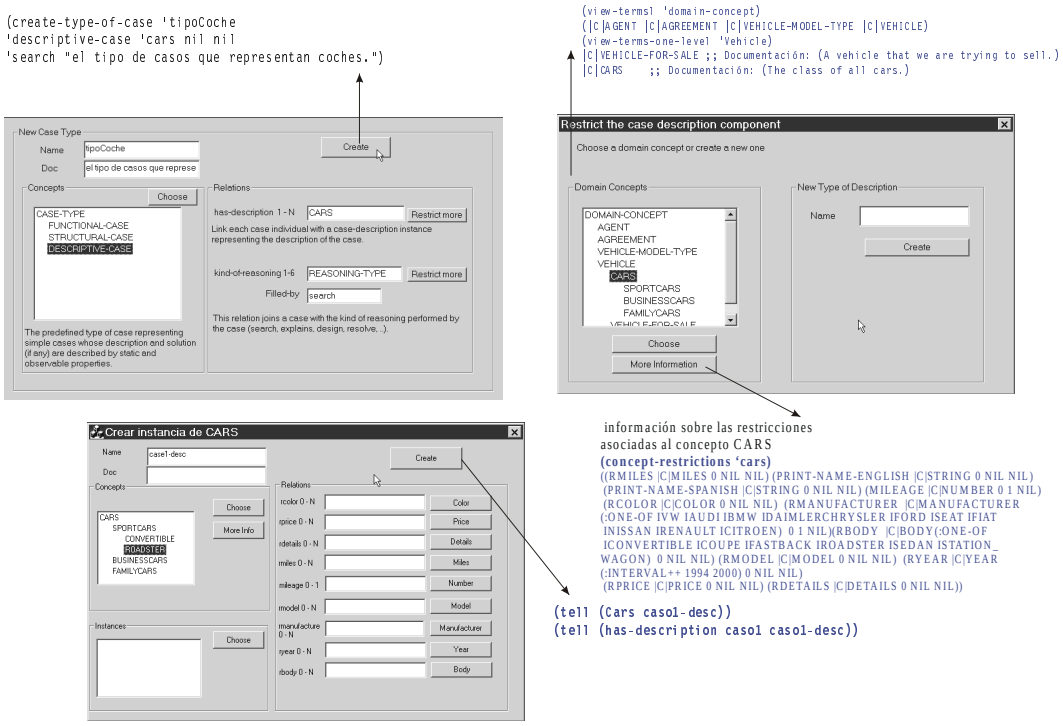


Figura 6-7. *Interfaz gráfica de COLIBRI*

que el sistema aserta que `Ford1` es una instancia de `Roadster`. El siguiente paso es la elección de los rellenos de los atributos del caso, que están determinados por el concepto al que pertenece, en este ejemplo `Roadster`.

Los atributos de un concepto se obtienen a través de una llamada a una función de la API de COLIBRI –función `concept-restrictions` (Apéndice D)– cuya respuesta incluye para cada atributo del concepto (heredado o específico) las siguientes propiedades: el nombre del atributo (role o relación), el rango, las restricciones de cardinalidad, mínimo y máximo, y el valor por defecto. En el fragmento de la Figura 6-6 se muestra como el sistema obtiene las instancias del concepto `COLORS` de las que el usuario elige una de ellas. Además como el atributo incluye como restricción de cardinalidad mínima el valor 1, el diálogo de la interfaz indica que se debe elegir uno de los valores de la lista obligatoriamente.

En la Figura 6-7 se muestra un fragmento de la interfaz gráfica para definir casos, en la que el intercambio de información se produce a través de las llamadas similares a funciones de la API de COLIBRI.

3.3 Selección y configuración de tareas y métodos

Como ya hemos comentado, el objetivo de esta fase es seleccionar las tareas a resolver en cada ciclo CBR, así como seleccionar y configurar los métodos que resolverán estas tareas. Teniendo en cuenta que estamos diseñando un prototipo de aplicación CBR, COLIBRI proporciona un entorno en el que podemos estudiar el comportamiento de distintos métodos, o distintas configuraciones del mismo método, por ejemplo, distintas medidas de similitud para comparar los resultados o distintos tipos de índices. El Apéndice C facilita un listado textual

completo de la biblioteca de métodos de CBROnto que hemos descrito en el Capítulo 5 y que serán los métodos disponibles para un diseñador que utilice COLIBRI para diseñar una nueva aplicación CBR. Además, el Apéndice B (formalización de CBROnto) incluye las definiciones en LOOM de las instancias que representan a los métodos de la biblioteca.

El apartado siguiente explica cómo crear un ciclo completo de resolución de problemas. Un diseñador puede definir varios ciclos CBR, por ejemplo, para configurar la interacción de distintos tipos de usuarios, tipos de consultas o tipos de casos (Apartado 3.3.2). El Apartado 3.3.3 se encarga de describir el proceso de selección y configuración de los métodos que se utilizarán en cada uno de estos ciclos CBR para la resolución de tareas y subtareas.

3.3.1 Creación de un ciclo CBR completo

Como vimos en el Capítulo 5, CBROnto define individuos como representantes canónicos de los conceptos de las taxonomías de tareas y métodos. COLIBRI incluye una función en su API –`new_CBR_cycle`– cuyo objetivo es la construcción de un ciclo CBR completo. Para ello se hace una copia completa de todos los individuos canónicos que representan a todas las tareas y los métodos de CBROnto, con sus asociaciones correspondientes, es decir, la competencia de los métodos (`competence`), enlaces de las tareas que tengan métodos preseleccionados (`task_method`) y subtareas (`subtask`) derivadas por cada método.

La función devuelve como salida la copia del individuo canónico (`iCBR_Task`) que es la instancia del concepto `CBR_TASK` que representa la tarea de resolución de problemas. Es decir, la salida de la función es un indiviudo que representa el nuevo ciclo CBR creado (ct_i). Cada vez que el usuario final inicia un ciclo de resolución de una consulta, el sistema hará una llamada al resolutor de problemas con esta tarea –`Resolve (`ct_i`).`

Un diseñador puede definir varios ciclos CBR, por ejemplo, para configurar la interacción de distintos tipos de usuarios, tipos de consultas o tipos de casos. Para ello se usarán individuos distintos –ct_1, ct_2, ...,ct_n– que son "hermanos" del representante canónico `iCBR_Task` del concepto `CBR_TASK` que se asociarán a los conceptos que representan los tipos de usuario, de consultas o de casos.

Al principio del proceso de diseño no existen vínculos específicos –enlaces `task_method`– entre cada tarea y el método concreto que se utilizará para resolver la tarea en la aplicación diseñada. El diseñador establecerá estos vínculos seleccionando entre las opciones que le ofrece COLIBRI accediendo a todos los métodos cuya competencia sea adecuada, es decir, métodos que pueden resolver la tarea si cuentan con el conocimiento del dominio establecido por sus requisitos.

3.3.2 Definición de tipos de consultas y tipos de usuarios

Como ocurre con otros mecanismos ofrecidos por COLIBRI, la definición de los tipos de consulta es una opción –no una obligación– para el diseñador del sistema. Hay que tener en cuenta que, aunque su uso es útil, por ejemplo, para restringir la estructura de las consultas válidas y para distinguir el comportamiento dependiendo de las características de la consulta del usuario, limita la flexibilidad de uso del sistema.

Como vimos en el Capítulo 4, con la definición de los tipos de consultas el diseñador restringe la estructura de las consultas que se permiten en el sistema. En particular, si no se define ninguna especialización cualquier consulta es instancia del concepto `Query-Type`, que no incluye restricciones. Además, los tipos de consultas pueden (o no) definirse como subconceptos de alguno de los tipos de casos y heredar sus características. El usuario final que

interactúa con el sistema diseñado podrá elegir entre alguno de los tipos de consultas existentes (conceptos más específicos de la jerarquía de consultas) y el sistema le ayudará a formular un individuo que satisfaga sus restricciones. Otra posibilidad es permitir la formulación de una consulta de forma libre y dejar que el sistema, usando el mecanismo de reconocimiento de instancias, compruebe la validez de la misma respecto a alguno de los tipos de consulta establecidos durante el diseño del sistema.

Otra opción de la que dispone el diseñador de la aplicación es la definición de distintos tipos de usuarios. Cada tipo de usuario se representará como un subconcepto del concepto `User_Types` de CBROnto. Este mecanismo permite que el diseñador pueda asociar distintas configuraciones de métodos, en concreto lo que se asocia son instancias de tareas, para cada tipo de usuarios del sistema. Teniendo en cuenta que cada instancia distinta de `CBR_TASK` representa un ciclo de resolución de problemas –que se ha creado con una llamada a la función `new_CBR_cycle`–, se pueden configurar ciclos (configurando los métodos que resuelven las tareas del mismo) que correspondan a usuarios más o menos expertos o con distintos grados de participación en los métodos.

Los tipos de consultas también pueden tener asignados ciclos distintos. Cuando comienza un ciclo de interacción con un usuario de la aplicación, es decir, la resolución de un problema con CBR, COLIBRI recupera la instancia de `CBR_TASK` adecuada dependiendo, en primer lugar del tipo de usuario:

- Cuando existe se aplica el ciclo asociado con el tipo de usuario actual y no se tiene en cuenta el ciclo asociado al tipo de consulta, ni a los tipos de casos.
- Si no existen tipos de usuario, o no tienen tareas asociadas (ni siquiera el concepto raíz `User_Types`) el sistema elige la instancia de `CBR_TASK` asociada al tipo de consulta actual, o al tipo de consulta más específico al que pertenece la consulta actual (hasta `Query-Type`) y que tenga asociado un ciclo CBR.
- Por último el sistema recupera y utiliza la instancia de `CBR_TASK` que esté asociada al tipo de caso actual, que representa qué tipo de casos queremos considerar en esta interacción y está definido por el atributo `casebase` del contexto actual. El Apartado 3.3.3 describe qué atributos se usan para la descripción del contexto de aplicación.

En un sistema simple, en el que se quiere configurar un único tipo de interacción independientemente del tipo de usuario, de consulta y de casos, el diseñador deberá asociar un único ciclo (instancia de `CBR_TASK`) a los conceptos `User_Types`, `Query-Type` o `CASE`, indistintamente –a través de la interfaz que hará una llamada a la función de la API `associate-cycle`.

Además de los tipos de usuario que defina el diseñador existe un tipo de usuario predefinido que describe al propio diseñador y que tiene asignadas las tareas de diseño. Este tipo de usuario es el único que puede resolver tareas que no representan ciclos de resolución de problemas (instancias de `CBR_TASK`), como las tareas de mantenimiento, de adquisición de casos, de integración de conocimiento, etc. Los usuarios finales del sistema sólo resuelven las instancias de `CBR_TASK` que correspondan al comportamiento configurado.

3.3.3 Selección y configuración de métodos

En esta fase del proceso de diseño de aplicaciones, el diseñador utilizará la interfaz para crear uno o varios ciclos CBR, elegir qué tareas se resolverán en cada uno y qué métodos se usarán para resolver cada una de esas tareas y sus subtareas, es decir, establecer los enlaces `task_method` –a través de la función de la API `link_task_method`.

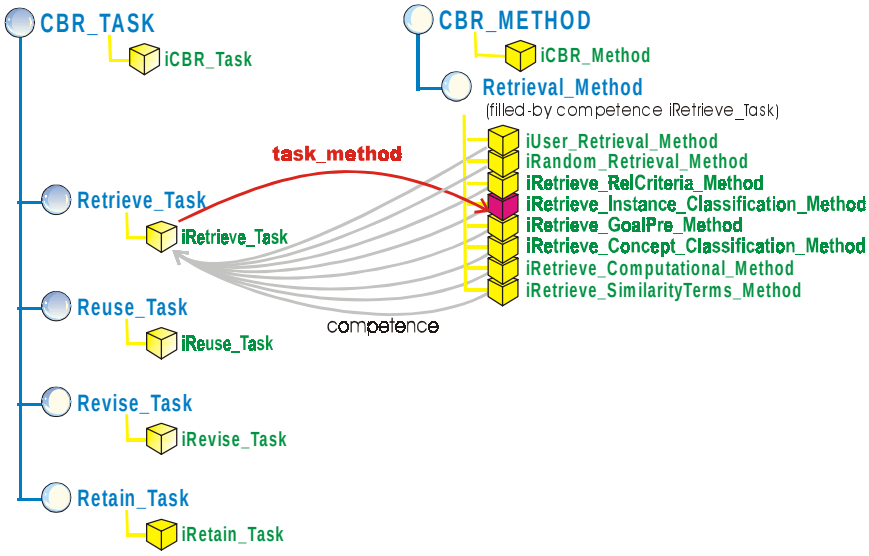


Figura 6-8. *Elección del método que resolverá una tarea*

En CBROnto la componente terminológica es estática aunque no es así la componente asertiva que varía según las características del sistema CBR diseñado. Según las características especificadas por el diseñador a través de la interfaz de COLIBRI, el sistema creará o eliminará individuos y realizará sobre ellos los asertos correspondientes para obtener el comportamiento deseado. Los individuos de la componente asertiva son específicos y representan las características concretas de cada sistema CBR, en particular, las relaciones –`task_method`– de las tareas CBR con los individuos que representan cuáles son los métodos utilizados.

En un principio cada instancia de la jerarquía de tareas estará relacionada –a través de la relación inversa de la competencia de los métodos (relación `competence`)– con distintos métodos que son los que se ofrecen como alternativas para resolver esta tarea. Por ejemplo, en la Figura 6-8 se muestran los métodos que se ofrecen para resolver la tarea de recuperación (individuo canónico `iRetrieve_Task`). COLIBRI –basándose en la competencia y en la aplicabilidad de los métodos en el contexto actual– ofrecerá al diseñador los métodos que existen para cada tarea a resolver, de los que el diseñador selecciona y configura aquellos que serán ofrecidos a los usuarios finales de la aplicación. Como ya hemos dicho, esta selección se explicita mediante asertos de la relación `task_method` entre la tarea y los métodos que el diseñador haya seleccionado. Estos asertos deben mantener las restricciones establecidas en la ontología. Por ejemplo, una tarea que es instancia de RETRIEVE_TASK debe relacionarse con un método que es instancia de RETRIEVAL_METHOD.

Normalmente para cada tarea el diseñador seleccionará y configurará un *único* método (un único enlace `task_method`) aunque, dependiendo del grado de participación que se quiera dar al usuario final, puede seleccionar más de uno (varios enlaces `task_method`) cuando quiera que en la aplicación el usuario final pueda elegir entre varios métodos distintos para resolver una misma tarea, o varias configuraciones del mismo método –es decir, el método será el mismo pero con requisitos de diseño del método distintos (esto se implementa con individuos distintos que son instancias del mismo concepto).

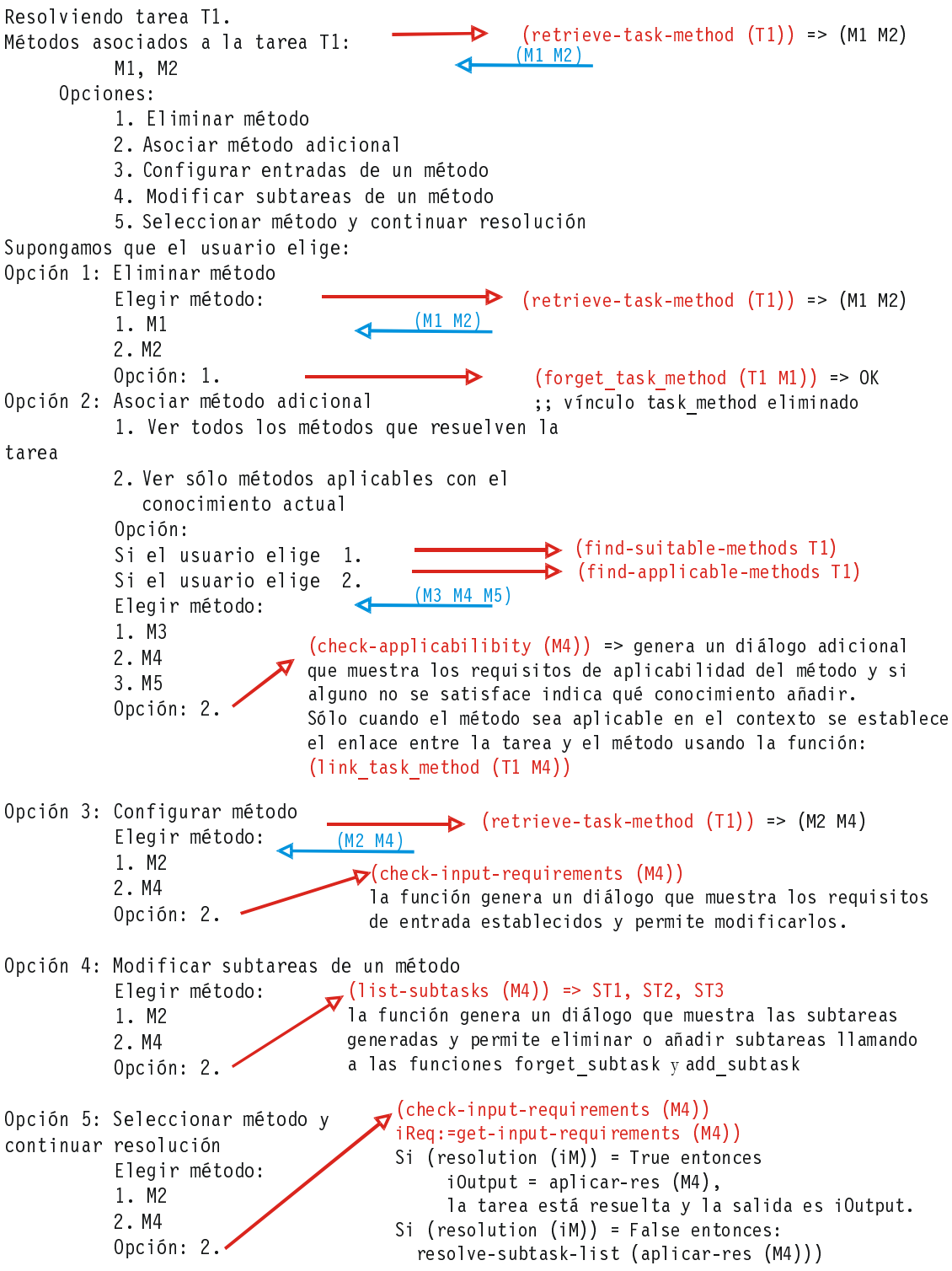


Figura 6-9. *Interfaz textual de selección y configuración de métodos*

La Figura 6-9 muestra un ejemplo de la interfaz textual de selección y configuración de métodos que se genera al iniciar el proceso de resolución de la tarea principal del ciclo CBR (instancia de `CBR_TASK`). En general, los pasos que lleva a cabo el resolutor de tareas durante la fase de diseño para resolver cualquier tarea `iT` son los siguientes:

1. Acceder a los métodos vinculados a la tarea `iT` a través de su relación `task_method` y dar opción al diseñador para:
   - Eliminar alguno de los *vínculos* entre la tarea y los métodos que estén seleccionados para su uso en la resolución de la tarea –los enlaces `task_method` se eliminan a través de la función de la API `forget_task_method`.
   - Seleccionar un método para resolver esta tarea, es decir, añadir un *vínculo* entre la tarea y un método –para establecer este enlace se usa la función de la API `link_task_method`. Los métodos que COLIBRI ofrece se obtienen de la aplicación del proceso de selección de métodos en función de su competencia y requisitos de aplicabilidad que se describe en el Apartado 3.3.3.1.
   - Cambiar la configuración de alguno de los métodos seleccionados para la tarea actual. El proceso de configuración de los requisitos de entrada de un método se describe en el Apartado 3.3.3.2.
   - Ver las subtareas que genera alguno de los métodos de descomposición (`list_subtasks`) para añadir o eliminar alguna de ellas (usando las funciones de la API `forget_subtask` y `add_subtask`).
   - Seleccionar alguno de los métodos `iM` para continuar el proceso de configuración. Además el diseñador puede añadir documentación textual al método (usando la función `put-documentation`) que se mostrará al usuario de la aplicación final y que puede ser útil para ayudarle a seleccionar un método si el diseñador asocia varios métodos con una tarea.
2. Con el método elegido `iM` simular la resolución de la tarea `iT`:
   - Acceder a los requisitos de entrada `iReq` (función `check-input-requirements`) y comprobar que el método esté configurado. Si no lo está dar opción a configurarlo.
   - Si `iM` es un método de resolución –se usa la función `resolution (iM)` que comprueba si `iM` es instancia de `Resolution_Method`– no se hace nada ya que durante el proceso de diseño sólo se simula la resolución de las tareas. Durante la resolución real de tareas en la aplicación final se aplica la especificación operacional del método.
   - Si `iM` es un método de descomposición (instancia de `Decomposition_Method`), el sistema hace una llamada a la función `aplicar-desc` que devuelve la secuencia de tareas (ST1, .. , STn) que se resuelven de forma recursiva:
     `ResolverSec (STn, ResolverSec (STn-1, ... (ResolverSec (ST3 (ResolverSec (ST2, Resolver(ST1))))) ..))`

     En la resolución de la secuencia de tareas, la salida del método que resuelve cada tarea se usa como la entrada del método que resuelve la siguiente tarea de la secuencia. La función `ResolverSec` comprueba que los requisitos de secuencia de los métodos se satisfacen y si encontrara algún problema informa al diseñador. Además, la función `ResolverSec` volverá al paso 1 para resolver cada subtarea generada.

3.3.3.1 Métodos aplicables que resuelven una tarea

Existen dos formas en las que COLIBRI recupera la lista de opciones de las que el diseñador de una aplicación CBR puede seleccionar qué método (o métodos) se vinculará a una cierta tarea. En la primera COLIBRI obtiene –función `find-suitable-methods`– todos los métodos que resuelven una tarea, usando la relación inversa de la competencia de los métodos, y el diseñador puede elegir de entre todos ellos. Después de la selección de un método el sistema comprueba si su aplicabilidad al contexto actual en función de sus requisitos, dando la posibilidad al diseñador para incluir el conocimiento adicional requerido por el método (en la fase de adquisición e integración de conocimiento del proceso de diseño de aplicaciones). En la segunda forma el proceso es similar –función `find-applicable-methods`– pero el sistema obtiene únicamente los métodos que son aplicables de entre todos los que resuelven la tarea, de forma que el propio contexto de aplicación determina los métodos adecuados para el tipo de conocimiento que está actualmente representado [Díaz&González02].

La representación explícita y declarativa de los requisitos de aplicabilidad de los métodos hace que sea posible razonar con ellos para comprobar si un método es o no adecuado para una cierta situación o contexto. Esta comprobación supone que el contexto de aplicación actual también se representa de forma declarativa. Para ello el sistema construye una descripción de las características de dicho contexto actual, incluyendo distintos tipos de información por ejemplo: la profundidad y anchura de sus taxonomías; el tipo de conocimiento que contiene, por ejemplo, el número de niveles terminológicos bajo algunos de los términos de CBROnto o su número de instancias –`Goal`, `Precondition`, `Problem_Type`, `Similarity_Measure`, `depends_on`; la estructura de organización bajo ciertos conceptos –`Case`, `Goal`, `Precondition`–; o la estructura de ciertos individuos. Esta descripción permite, en general, determinar el grado de integración del conocimiento del dominio y el de CBROnto.

Para caracterizar el contexto actual, COLIBRI construirá un individuo usando ciertos descriptores (atributos o relaciones) que lo definen. Los descriptores del contexto son computados por el sistema en función de las características de la base de conocimiento. Este individuo se clasifica en la taxonomía de conceptos, en concreto en la parte de cualificadores de conocimiento (subconceptos de `Knowledge-Qualifiers`), y se comprueba si el concepto de requisitos asociado al método elegido subsume al individuo que representa el contexto actual. Si es así, el método es aplicable y si no el método no es aplicable. Además, este mecanismo hace posible que COLIBRI explique por qué no es adecuado y qué conocimiento se debería incluir para que el método fuese aplicable.

Como ejemplo muy sencillo para ilustrar este proceso, el concepto `Current_Context` de la Figura 6-10 representa una situación en la que disponemos de una taxonomía de conceptos profunda (`With_Domain_Concept_Taxonomy_Deep`) aunque de poca anchura (`With_Domain_Concept_Taxonomy_Narrow`), la jerarquía de relaciones del dominio sólo tiene un nivel (`dom-rt-depth 1`) y la base de casos tiene tamaño medio (`With_Case_Base_medium_size`). Suponemos que queremos comprobar si el método de recuperación por reconocimiento de instancias es aplicable. Recordamos que los requisitos de aplicabilidad del método establecen que el modelo de conocimiento del dominio debe estar suficientemente poblado, es decir, jerarquías de conceptos y relaciones de altura y anchura media.

La Figura 6-10 muestra un ejemplo del mecanismo para determinar si el método es o no aplicable, que se basa en el mecanismo de subsunción y en la representación explícita de una jerarquía de conceptos (subconceptos de `Knowledge-Qualifiers`) que describen el conocimiento existente. El individuo contexto se clasifica en la posición que le corresponda en la jerarquía (en función de sus descriptores), y se comprueba si el concepto de requisitos del

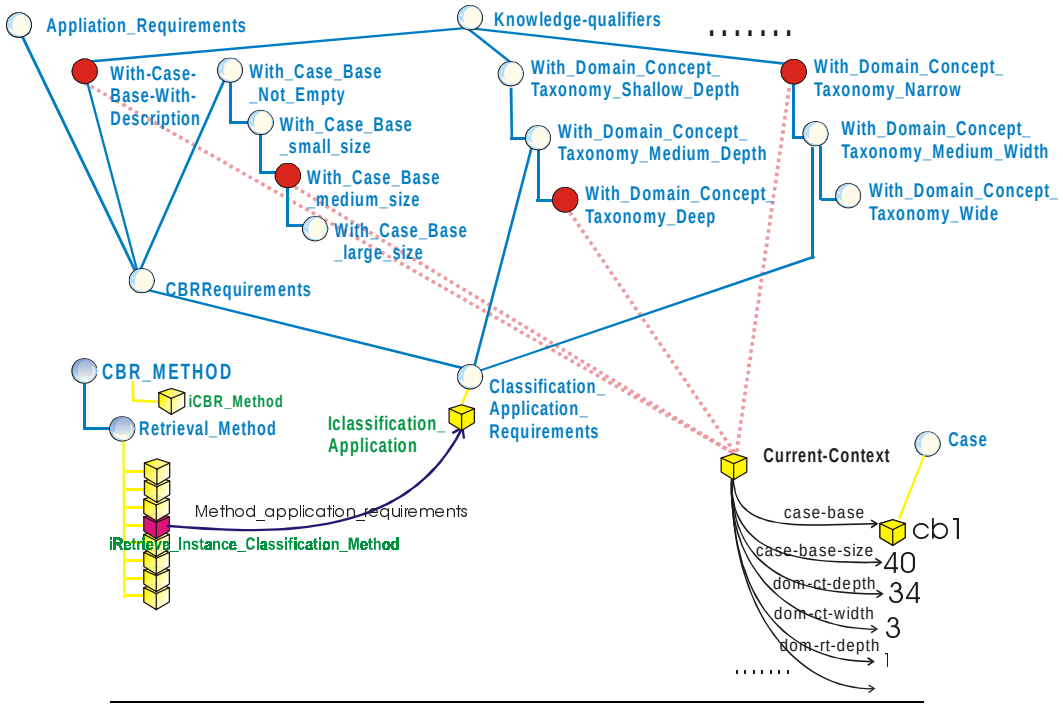


Figura 6-10. *Aplicabilidad de un método respecto al contexto*

método lo reconoce como una instancia suya. En el ejemplo, el método no es aplicable porque requiere una taxonomía de conceptos de anchura y profundidad media, mientras que el contexto actual representa una situación en la que la taxonomía de conceptos es profunda (y por subsunción es adecuada) pero estrecha, con lo que no satisface los requisitos del método. El concepto `With_Domain_Concept_Taxonomy_Narrow` no subsume al concepto `With_Domain_Concept_Taxonomy_Medium_Width` y, por tanto, el concepto de requisitos del método (`Classification_Application_Requirements`) no reconoce como instancia suya al individuo que representa el contexto actual.

Aunque el método no es aplicable la representación declarativa permite que COLIBRI explique cuál es la necesidad de conocimiento del método que no se satisface y poder resolver la carencia. En este ejemplo, el sistema puede indicar que para que el método sea aplicable es necesario poblar en anchura la jerarquía de conceptos del dominio.

El siguiente apartado describe los atributos que utiliza COLIBRI como descriptores del individuo que representa el contexto actual, así como la jerarquía de conceptos que representan los cualificadores de conocimiento (que se muestra en la Figura 6-11).

Descriptores del contexto y cualificadores del conocimiento

Para describir la situación o contexto actual en el que se aplica un cierto método, COLIBRI construye un individuo usando alguno de los siguientes descriptores (relaciones):

- `casebase:` individuo que representa a la base de casos sobre la que quiere restringir la aplicación de los métodos (por defecto es `CASE` con lo que trabajamos con todos los casos).
- `query:` individuo que representa la consulta actual (si existe).
- `case-base-size`: número de casos en la base de casos seleccionada en el atributo casebase.

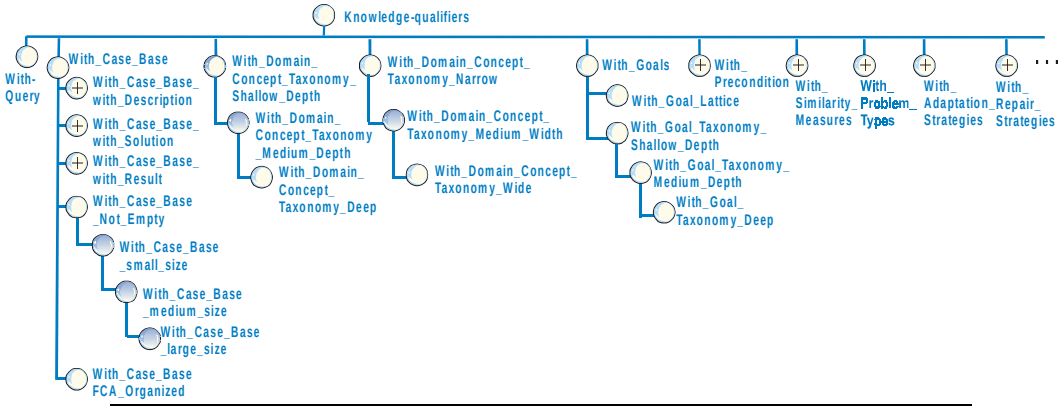


Figura 6-11. *Cualificadotes de conocimiento*

- `domain-concept-taxonomy-width` y `domain-relation-taxonomy-width`: anchura de las taxonomías de conceptos y relaciones del dominio, respectivamente.
- `domain-concept-taxonomy-depth` y `domain-relation-taxonomy-depth`: profundidad de la taxonomía de conceptos y relaciones del dominio, respectivamente.
- `Focus-concept`: referencia al individuo que representa el concepto del dominio bajo y sobre el cual me interesa comprobar la profundidad de las jerarquías. Se utiliza para el método de recuperación por clasificación. Este concepto (focus concept) es el destino de la cadena de relaciones que indica el individuo (la parte del caso) que hay que clasificar.
- `Focus-concept-taxonomy-depth`: profundidad de la taxonomía del dominio bajo el concepto representado en el atributo `focus-concept`.
- `goal-depth`, `precondition-depth`, `Adaptation-Strategy depth`, `Problem-Types depth`, `Fail-Type-depth`: profundidad de la taxonomía del dominio bajo los conceptos `Goal`, `Precondition`, `Adaptation-Strategy`, `Problem-Types` y `Fail-Type`, respectivamente.
- `goal-organization-structure`, `pre-organization-structure`, `case-organization-structure`, `case-base-organization-structure`: estructura de los conceptos clasificados bajo `Goal`, `Precondition`, `Case` o el concepto representado por el relleno del atributo `case-base` del contexto. Será uno de los dos individuos: `FCA_lattice` o `none`.
- `SimilarityMeasure_depth`: número de medidas de similitud definidas, es decir, número de instancias del concepto `Similarity_Measure`.
- `Depends_on_depth`, `Depends_on_uses`: número de relaciones clasificadas bajo la relación `depends_on` y número de veces que se usa cualquiera de ellas (incluyendo `depends_on`) en la ABox.

Como ya hemos explicado, COLIBRI calcula automáticamente los valores para los atributos anteriores. En función de estos valores el individuo que representa el contexto actual se clasifica en la jerarquía de cualificadores de conocimiento (subconceptos de `Knowledge-Qualifiers`) que se muestra en la Figura 6-11, sobre la que también están clasificados los conceptos que representan los requisitos de aplicabilidad de los métodos de CBROnto (como en la Figura 6-10). Esta clasificación permite determinar si un cierto método es aplicable en un cierto contexto según los requisitos de aplicabilidad que hemos visto en el Capítulo 5 para cada uno de los métodos. La definición en LOOM de los conceptos de la

para cada uno de los métodos. La definición en LOOM de los conceptos de la figura se puede consultar en el Apéndice B.

3.3.3.2 Configuración de métodos

La configuración de los métodos se refiere a la especificación de sus requisitos de entrada, en concreto, de los requisitos paramétricos, de diseño y de conocimiento, ya que los de secuencia no se pueden modificar. De esta labor se encarga la función de la API `check_input_requirements`, que genera un diálogo en el que muestra los requisitos de entrada del método, tanto los que estén ya establecidos como los que no, y permite hacer modificaciones en ellos.

La función recibe un método `iM` y, usando la relación `method_input`, accede a la instancia de requisitos de entrada `iReq`, que aúna los requisitos de diseño (relación `design_requirements`), los de conocimiento (relación `knowledge_requirements`) y los paramétricos (`parameter_requirements`).

Durante la fase de diseño de la aplicación se deben establecer los requisitos de diseño y de conocimiento. Los requisitos de diseño se especifican a través de la función de la API `put-design-requirements` que recibe un requisito (atributo del individuo de requisitos de diseño del método) y el relleno correspondiente. En el caso de los requisitos de conocimiento, el diseñador tendrá que usar funciones específicas de la API, como `createSimilarityFunction`, `createSimilarityMeasure`, `associateSimilarityMeasuretoConcept`, `create-relevance-criteria`, `add-suconcept`, etc.

El diálogo de interfaz proporciona ayuda en ambos casos indicando qué atributos o requisitos hay que rellenar y el tipo de valores, en la configuración de los requisitos de diseño, y qué funciones hay que utilizar para configurar los requisitos de conocimiento. Veremos un ejemplo de uso de estas funciones en el ejemplo del Apartado 4.

3.3.4 Ejemplo de configuración de métodos

Como diseñadores del sistema CBR de venta de coches, en esta fase del proceso de diseño configuramos las tareas y los métodos de la aplicación. Decidimos la opción más simple, un único ciclo CBR para los dos tipos de casos.

A través de la interfaz se harán las llamadas a la función de la API de creación de un nuevo ciclo CBR (que se asociará el concepto `CASE`) y a la función de comienzo de la resolución de la tarea lo que genera la interfaz textual de selección y configuración de métodos que se muestra en la Figura 6-12. En la interacción el usuario *elimina* las tareas de reutilización, revisión y aprendizaje del método CBR, es decir, la única tarea que se resuelve es la recuperación, y asocia a la misma el método de recuperación por cómputo de similitud. Realmente el resto de las tareas quedan vinculadas con el método `ido_nothing_Method`, con lo que el efecto es la no resolución de las mismas en este ciclo.

Posteriormente habrá que configurar los requisitos de entrada del método. En particular, los requisitos de conocimiento del método de recuperación por cómputo de similitud sugieren la definición de medidas de similitud asociadas a los conceptos del dominio, en particular a los conceptos `Car`, `Purchasing-Agreement` y `Color` con lo que el diseñador procede a su creación. Tras este paso de configuración la aplicación queda lista para la fase de pruebas.

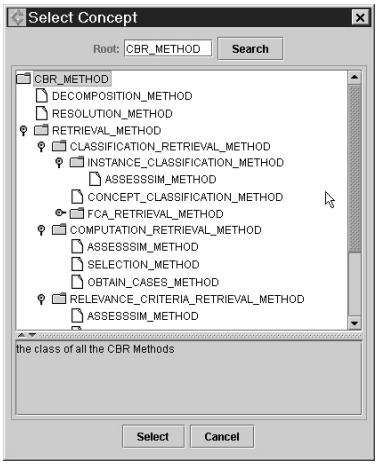
```
Resolviendo tarea "principal CBR" (icbr_task2).
Métodos asociados a la tarea "principal CBR":
      Método CBR (método principal CBR)
      Opciones:
            1. Eliminar método
            2. Asociar método adicional
            3. Configurar entradas de un método
            4. Modificar subtareas de un método
            5. Seleccionar método y continuar resolución

Opción: 4
      Método CBR ¿Modificar subtareas S/N? S
            1. Eliminar tareas
            2. Añadir nuevas tareas
            3. Salir
            Opción: 1
            Subtareas actuales:
            1.Recuperación de casos
            2.Reutilización de casos
            3.Revisión de casos
            4.Aprendizaje
            Eliminar? (0 para salir): 2,3,4  0
(Volvemos al menú principal)
Opción 5: Seleccionar método y continuar resolución
            Método CBR ¿Seleccionar S/N? S

Resolviendo tarea "Recuperación de casos" (iretrieve_task2)
Métodos asociados a la tarea "Recuperación de casos"
      Ningún método asociado actualmente
      Opciones:
            1. Eliminar método
            2. Asociar método adicional
            3. Configurar entradas de un método
            4. Modificar subtareas de un método
            5. Seleccionar método y continuar resolución
      Opción: 2
            1. Ver todos los métodos que resuelven la tarea
            2. Ver sólo métodos aplicables con el
               conocimiento actual
            Opción: 1
            Elegir método:
            1. Computacional " computa la similitud .."
            2. Clasificación  " basado en clasificar .."
            3. Términos de similitud
               .......
            Opción: 1.

Resolviendo tarea "Recuperación de casos" (iretrieve_task2).
Métodos asociados a la tarea "Recuperación de casos"
      Computacional
      Opciones:
            1. Eliminar método
            2. Asociar método adicional
            3. Configurar entradas de un método
            4. Modificar subtareas de un método
            5. Seleccionar método y continuar resolución
       Opción: 3
```

```
(retrieve-task-method (icbr_task2))
                              => (icbr_method2)
(print-name (icbr_method2))
=> "Método CBR"
(get-documentation (icbr_method2))
=> "Método principal CBR"
```

```
(("Método CBR" "Método principal CBR"))
```

```
(list-subtasks (icbr_method2))
 => (iretrieve_task2, ireuse_task2,
    irevise_task2, iretain_task2)
(print-name (iretrieve_task2))
=> "recuperación de casos"
(print-name (ireuse_task2))
=> "reutilización de casos"
..
```

```
(forget_task_method (ireuse_task2 icbr_method2))
(forget_task_method (irevise_task2 icbr_method2))
(forget_task_method (iretain_task2 icbr_method2))
```




```
(find-suitable-methods (iretrieve_task2))
```

```
(check-applicabilibity (icomputational_method2))
;; comprueba que el método es aplicable al contexto
(link_task_method (iretrieve_task2 icomputational_method2))
```

```
(check-input-requirements (icomputational_method2)))
la función genera un diálogo que muestra los requisitos
de entrada establecidos y permite modificarlos.
```

Figura 6-12. *Resolución de tareas*

3.4 Depuración y pruebas

Tras la resolución de las tareas de diseño de la aplicación, cada ciclo de resolución de problemas –o ciclo de interacción con el usuario– se corresponde con la resolución de alguna de las instancias de CBR_TASK, que son las representantes de los ciclos de resolución de problemas que se han configurado en la aplicación. Si no se ha hecho anteriormente, el diseñador debe asociar cada una de las instancias de CBR_TASK con los conceptos que representan los tipos de usuario, tipos de casos o tipos de consulta, dependiendo del uso de cada ciclo de interacción –usando las funciones de la API `associate-cycle y free-cycle.`

En esta fase de depuración y pruebas, el diseñador puede probar los métodos configurados utilizando el mismo proceso de resolución de tareas que se usará en la aplicación final y que es similar al que hemos descrito para el proceso de diseño, salvo que no se permiten modificaciones en las tareas que deriva un método, ni en los requisitos de diseño ni de conocimiento. Además, en la aplicación final los enlaces entre las tareas y los métodos y los requisitos de diseño estarán fijados.

En cuanto a la configuración de tareas y métodos la intervención del usuario final en la aplicación consistirá en:

- Elegir entre los métodos alternativos asociados a una tarea. Esta elección sólo es necesaria si existen varios enlaces `task_method`. Si la tarea está vinculada con un único método, será éste el que se utilice.
- Configurar los requisitos paramétricos de los métodos utilizando la ayuda del sistema (los requisitos se establecen usando la función de la API `put-parameter-requirements).`

COLIBRI define un proceso de resolución de tareas general y recursivo que parte de una tarea que queremos resolver y accede a los métodos elegidos y configurados por el diseñador (usando la función de la API `retrieve_task_method`). Si hay más de un método, será el usuario el que elija uno de ellos usando como ayuda la documentación que el diseñador haya incluido en el método. Si el método es de resolución se usa directamente para resolver la tarea y si no la divide en subtareas y para cada una de ellas aplicamos el proceso de resolución de forma recursiva.

3.4.1 Resolución de tareas en la aplicación final

El proceso de resolución de tareas en la aplicación final es similar al que hemos descrito en el Apartado 3.3.3 para el proceso de configuración de tareas. La diferencia fundamental es que como vimos, durante la fase de diseño las tareas no se resuelven sino que únicamente se simula el proceso de resolución que se lleva a cabo en la aplicación final.

Los pasos que lleva a cabo el resolutor de tareas para resolver cualquier tarea `iT` son:

1. Acceder a los métodos vinculados a la tarea `iT` a través de su relación `task_method.` Si hay varios se dará opción al usuario para elegir uno de ellos: `iM`.
2. Con el método `iM` resolver la tarea `iT`:
   - Acceder a los requisitos paramétricos y dar opción a configurar el método.
   - Si `iM` es un método de resolución utilizarlo para resolver `iT`. El sistema hace una llamada a la función `aplicar-res` que accede a la especificación operacional del método (relación `operational_specification` de `iM`) y aplica la función Lisp usando los requisitos de entrada `iReq`, para obtener el resultado `iOutput`.

– Si `iM` es un método de descomposición (instancia de `Decomposition_Method`), el sistema hace una llamada a la función `aplicar-desc` que devuelve la secuencia de tareas (ST1, .. , STn) que se resuelven de forma recursiva (vuelta al paso 1). En la resolución de la secuencia de tareas, la salida del método que resuelve cada tarea se usa como la entrada del método que resuelve la siguiente tarea de la secuencia.

El siguiente apartado ejemplifica el uso de COLIBRI y las fases de diseño para una aplicación más compleja.

4. Ejemplo: Generación de Poesía

Como ejemplo de uso de COLIBRI y CBROnto, en este apartado describimos el diseño de un sistema CBR que genera poesías en castellano tomando como punto de partida un texto dado por el usuario y una base de casos de poemas previos.

En general, la labor de generación automática de textos radica en encontrar estructuras lingüísticas correctas que representen un mensaje que se quiere transmitir. En este ejemplo nos centramos en textos poéticos que se atienen a formas estróficas y a estructuras métricas tradicionales [Quilis85]. La aplicación no pretende ser un generador automático de poemas que compita con poetas reales, ni reducir el proceso de creación literaria a un conjunto de transformaciones básicas sobre poemas previos, aunque sí estudia aquella parte de los procesos involucrados que es susceptible de cierta mecanización. Por ejemplo, el número de sílabas impone restricciones en el tipo de versos que se pueden generar. Por otro lado, las reglas de la forma estrófica a utilizar imponen restricciones a la estructura general del poema.

Existen versiones previas de esta aplicación utilizando distintas tecnologías: WASP, ASPID [Gervás00] [Gervás01a]. En este apartado estudiamos la adecuación, ventajas y posibilidades de COLIBRI para esta aplicación. En un primer análisis se observan características del dominio y de la aplicación que resultan adecuadas para los métodos de CBROnto. Por ejemplo, se puede sacar partido de una representación explícita del modelo del dominio, las características que describen las reglas de la poesía formal se pueden expresar mediante definiciones terminológicas que permiten comprobar la *corrección* de un poema generado, los casos son complejos y se pueden identificar distintos tipos de casos con distintos niveles de complejidad.

En los siguientes apartados describimos el diseño del generador de poemas usando COLIBRI/CBROnto. Siguiendo las fases descritas en el apartado anterior, el diseñador de la aplicación llevará a cabo una fase de modelado del dominio e integración con CBROnto, la definición de la base de casos, de los tipos de consulta y de usuarios y la selección y configuración de los métodos CBR.

4.1 Adquisición de conocimiento del dominio

Aunque hubiese sido deseable, no hemos encontrado ontologías del dominio de la aplicación planteada –la poesía formal en castellano– que pudiéramos reutilizar. Por tanto, hemos tenido que hacer el esfuerzo inicial de modelar una base de conocimiento sobre el dominio, esperando que el resultado pueda ser reutilizado en otras aplicaciones, por ejemplo añadiéndolo como ontología a alguna de las bibliotecas existentes.

El objetivo de este apartado es describir los aspectos de la métrica tradicional que hemos formalizado mediante una representación terminológica y cómo se han modelado usando

LOOM aprovechando sus mecanismos de razonamiento –principalmente la detección de incoherencias y el reconocimiento de instancias. En el Apéndice E se incluye la formalización completa del modelo del dominio.

4.1.1 Reglas básicas de la poesía en castellano

Existen ciertas reglas de la poesía formal en castellano para determinar el número de sílabas métricas, el tipo de verso o la forma estrófica que se usa en un cierto poema. Sabiendo que cada palabra se divide en sílabas y que sólo una de ellas lleva el acento prosódico, las reglas que restringen la validez de un poema tienen en cuenta los siguientes aspectos:

Número de sílabas métricas. Una de las características principales de los versos, respecto a la métrica de la estrofa que forman, es su número de sílabas. En general, la sílaba métrica no siempre coincide con la sílaba morfológica, ya que cuando una palabra termina en vocal y la siguiente palabra empieza en vocal, la última sílaba de la primera palabra y la primera sílaba de la siguiente palabra forman una única sílaba métrica. Este fenómeno se conoce como *sinalefa* e influye en el problema de establecer la cuenta de sílabas de un verso. Por ejemplo, el verso "*bástete amor lo que ha por mí pasado*" tiene 13 sílabas morfológicas pero tiene 11 sílabas métricas ya que tiene dos casos de sinalefa (marcados en negrita): "*bástete* **-** *amor lo qu*e **-** *ha por mí pasado*".

Formas estróficas. Aunque un poema puede ser una secuencia de versos sin estructurar y con muy pocas restricciones, en esta aplicación estamos interesados específicamente en la generación de poemas que se ciñen a formas estróficas concretas, por ejemplo, sonetos, tercetos o cuartetos. Cuando generamos un nuevo poema, las restricciones impuestas por la forma estrófica escogida pueden afectar a distintos parámetros:

- Pueden imponer una cierta restricción al **número de versos** del poema. Por ejemplo, un cuarteto tiene exclusivamente cuatro versos.
- Pueden determinar la longitud de los versos (**número de sílabas métricas**): por ejemplo, un romance tiene versos de ocho sílabas.
- Pueden determinar la **rima** de algunos o todos los versos como en los tercetos, en los que rima el primer verso con el tercero, o en los cuartetos, en los que riman los versos primero y cuarto, y segundo y tercero.

El poeta que genera un poema no necesariamente tiene presente estas restricciones en forma de reglas, sino que ha desarrollado una sensibilidad (de cuya formalización concreta en la mayoría de los casos el poeta mismo no es consciente) que le permite distinguir cuando un borrador o una idea son poéticamente aceptables. En WASP y ASPID se define una aproximación formal de esta sensibilidad en términos del número y naturaleza de las palabras que constituyen un verso. En nuestra aplicación existe un modelo terminológico del dominio que guía la representación de casos y se define una aproximación basada en casos de esta sensibilidad, ya que los casos (poemas) previos cumplen las reglas formales. Al generar un nuevo poema basado en uno anterior se mantienen los aspectos formales (sintácticos y métricos) adaptando, mediante sustitución local, las palabras concretas que se utilizan. Es decir, configuraremos una aplicación en la que se generan poemas manteniendo la estructura de un poema recuperado y sustituyendo sus palabras por las palabras dadas en la consulta, manteniendo las características sintácticas y métricas del poema resultante.

4.1.2 Formalización del modelo del dominio

En el modelo del dominio los poemas se representan como instancias del concepto primitivo `Poema_Concept`. Cada poema se relaciona con sus estrofas –que se representan como instancias del concepto `Estrofa_Concept`–, y cada estrofa se relaciona con sus versos –instancias del concepto `Verso_Concept`–, que es un tipo de línea de texto (subconcepto de `Línea_Texto`) para el que, además de sus palabras –relación `tiene-palabra`–, se representa su rima y número de sílabas –a través de las relaciones `rima-verso y numero-silabas-verso`, respectivamente. Además, un verso de una estrofa puede estar *encabalgado* con el siguiente verso para indicar que forman una única oración.

El número de sílabas de un verso se obtiene usando una función que computa la suma del número de sílabas de las palabras del verso teniendo en cuenta las sinalefas. En función del valor computado para este atributo en cada instancia del concepto `Verso_Concept`, el sistema clasifica los versos en `Heptasílabos`, `Octosílabos`, `Eneasílabos`, `Decasílabos`, `Endecasílabos` y `Dodecasílabos`. Las definiciones de los conceptos se pueden consultar en el Apéndice E.

Los distintos tipos de estrofas se representan mediante una jerarquía de subconceptos de `Estrofa_Concept` que clasifican las estrofas en tres tipos: romances, tercetos y cuartetos. Un poema puede estar formado por una única estrofa –como los romances, tercetos y cuartetos– o por varias –tercetos encadenados y sonetos:

- Los tercetos son las estrofas –instancias de `Estrofa_Concept`– que el sistema reconoce como pertenecientes al concepto `Terceto_Concept`. Se definen como aquellas estrofas con exactamente tres versos endecasílabos –instancias de `Endecasílabo`. En general los tercetos no tiene restricciones de rima, aunque distinguimos un tipo de tercetos característicos en los que el primer verso rima con el tercero (`TercetoUno-Tres` que es subconcepto de `Terceto_Concept`).
- Los cuartetos son las estrofas reconocidas como pertenecientes al concepto `Cuarteto_Concept`. Se definen como aquellas estrofas con cuatro versos `Endecasílabos`. Además una característica común de todos los cuartetos es su rima: el primer verso rima con el cuarto, y el segundo con el tercero.
- Los romances son las estrofas –instancias de `Romance_Concept`– que tienen un número par de versos `Octosílabos` y tales que riman los versos pares. La rima en este tipo de estrofas es asonante y no consonante como en los tipos anteriores. En la versión actual de la aplicación sólo tenemos en cuenta la rima consonante.
- Un terceto encadenado –instancia de `Terceto_Encadenado`– es un tipo de estrofa de seis versos `Endecasílabos` cuya rima es ABA BAB. En nuestro modelo del dominio un terceto encadenado no se define en términos de sus versos, sino como un tipo de poema que se compone de varias estrofas simples, en concreto dos tercetos que riman de forma inversa. Con esta aproximación, cada terceto es un caso simple que forma parte de otro caso más complejo (el terceto encadenado).
- Un soneto –instancia de `Soneto_Concept`– es una estrofa de 14 versos `Endecasílabos` cuya rima es del tipo ABBA ABBA CDE CDE o ABBA ABBA CDC DCD. Igual que en los tercetos encadenados, en nuestro modelo del dominio un soneto se define como un tipo de poema que se compone de varias estrofas simples, en concreto dos cuartetos y dos tercetos.

4.1.3 La representación del vocabulario

Una parte muy importante del modelo del dominio de la poesía formal en castellano es la representación del vocabulario, es decir, de las palabras que forman los versos. En esta aplicación trabajaremos con un vocabulario predefinido donde para cada palabra representamos: su rima consonante, el número de sílabas, si empieza o termina por vocal y en qué sílaba tiene el acento. El procesamiento de las palabras se realiza externamente mediante un analizador en Prolog que extrae la información adecuada (© P. Gervás) y un traductor que define las instancias LOOM y aserta sobre ellas la información extraída por el analizador. Cada palabra se representa como un individuo que es instancia del concepto del dominio `Palabra_Concept` y para cada una distinguimos los siguientes atributos (relaciones) con el significado intuitivo: `textoPalabra`, `categoria-sintactica`, `numero-silabas-palabra`, `acento`, `rima-palabra`, `empVocal` (empieza por vocal) y `terVocal` (termina por vocal).

Actualmente disponemos de 4872 palabras en el vocabulario, de las cuales 313 son las que aparecen en los casos y el resto es vocabulario adicional que está disponible para la generación de consultas y nuevos poemas. En el dominio, el conjunto de las palabras del vocabulario se clasifica automáticamente de diversas formas en función de sus características, es decir, de los valores de sus atributos. Por ejemplo, como hemos descrito, a efectos del cómputo de sílabas métricas de un verso es importante si las palabras terminan o empiezan por vocal. En el modelo del dominio definimos los conceptos `Termina_Vocal`, `No_Termina_Vocal`, `Empieza_Vocal`, `No_Empieza_Vocal`, que permite clasificar automáticamente las palabras de forma adecuada en cuanto a este aspecto. Además hemos definido otros conceptos como `Dos_Silabas`, `Tres_silabas`, `Cuatro_Silabas`, `Acento_Dos` o `Acento_Tres` que clasifican las palabras según su número de sílabas y la posición del acento en la palabra. Las definiciones se pueden consultar en el Apéndice E.

4.1.3.1 Las categorías sintácticas de las palabras

Cada una de las palabras del vocabulario de la aplicación está relacionada con un individuo que representa su categoría sintáctica. Para ello hemos utilizado las categorías del proyecto CRATER y su etiquetador (POS –*Part Of Speech*– *tagger*) [Sánchez-León93] que extrae de forma automática la categoría sintáctica de una palabra. Entre las etiquetas sintácticas identificadas se encuentran, por ejemplo:

ADJGFP Feminine plural general positive adjective

ADJGFS Feminine singular general positive adjective

ADJGMP Masculine plural general positive adjective

ADJGMS Masculine singular general positive adjective

Aunque el conjunto original de categorías es *plano*, es decir, las categorías no están explícitamente relacionadas entre sí, se puede observar que mantienen una relación implícita entre ellas que se refleja en su nombre. Por ejemplo, ADJGFP y ADJGFS son adjetivos (ADJ) generales (G) para los que distinguimos entre femenino (F), masculino (M), plural (P) y singular (S). En la aplicación hemos utilizado este hecho para construir, como parte del modelo del dominio, una taxonomía explícita de categorías sintácticas basada en las etiquetas originales, resultando una jerarquía de 544 conceptos en la que se clasifican los individuos que representan categorías sintácticas concretas con las que se relacionan las palabras del vocabulario. Es decir, la jerarquía no clasifica a los individuos que representan las palabras, sino que cada palabra se relaciona con un individuo `cs` que representa su categoría y es este individuo `cs` el que se clasifica en la jerarquía de categorías sintácticas. Algunos de estos conceptos se pueden consultar en el Apéndice E.

Consideramos que cada individuo palabra (instancia de `Palabra_Concept`) tiene una única categoría sintáctica. Para representar el hecho de que en castellano una palabra puede tener distintas categorías sintácticas, utilizaremos distintos individuos que comparten todos los atributos (número de sílabas, acento, rima, propiedades de empezar y terminar por vocal y texto de la palabra) salvo la categoría sintáctica. Por ejemplo, dependiendo del contexto en el que aparezca, la palabra "rosa" puede tener asociadas las categorías *NCFS (nombre común femenino singular), ADJGMS, ADJGFS (adjetivo general masculino- femenino singular)*, por lo que representaremos cinco instancias de palabra para formalizarla.

La representación de todas las categorías sintácticas de una palabra en el mismo individuo (instancia de `Palabra_Concept`) supondría que no se distingue la categoría concreta de una cierta aparición de la palabra en cuestión, con los consiguientes errores en las sustituciones basadas en esta propiedad durante la adaptación de los poemas.

4.1.3.2 Las apariciones de las palabras

En el modelo del dominio distinguimos entre las palabras (instancias de `Palabra_Concept`) y las apariciones de las palabras (instancias de `Aparicion_Palabra`). En la representación de los poemas utilizamos un individuo distinto para cada aparición de una palabra, que se relaciona con la palabra que representa (a través de la relación `de-palabra`) y con la aparición de palabra siguiente (a través de la relación `anterior-a-palabra`). De esta forma aislamos las palabras del vocabulario de las apariciones reales de las palabras en los versos de las estrofas de los poemas. Dos apariciones de la misma palabra serán muy similares (ya que comparten una relación `de-palabra` con el mismo individuo) aunque serán distintas en cuanto al verso en el que aparecen y a la posición que ocupan en él.

Dependiendo de la medida de similitud utilizada y del tipo de índice, que determina qué relaciones tener en cuenta y con qué importancia, dos apariciones de la misma palabra ambas ocupando la posición del final de un verso deberían ser más similares que si una de ellas aparece en el medio de un verso y la otra al final. Por ejemplo, el uso del tipo de índice temporal incluye a la relación `anterior-a-palabra` (clasificada bajo la relación `before`).

En el modelo del dominio las apariciones de las palabras se clasifican en una jerarquía conceptual (subconceptos de `Aparición_Palabra`) según distintos criterios:

- Distinguimos entre dos tipos de apariciones de palabras: las que son final de línea y las que no. De nuevo es el mecanismo de reconocimiento de instancias de LOOM el que se encarga de clasificar las apariciones de las palabras en uno u otro tipo según las definiciones de los conceptos `Palabra_Final_Linea` y `Palabra_No_Final_Linea`. Además, dentro de las apariciones que son final de línea distinguimos entre aquellas apariciones que requieren que se mantenga la rima (`Aparición_Palabra_Rimada`) y aquellas que no (`Aparición_Palabra_No_Rimada`). Por ejemplo, en la última palabra del primer y tercer verso de un soneto uno tres se debe mantener la rima pero en la última palabra del segundo verso no es necesario. La pertenencia de una aparición de palabra a uno u otro concepto es inferida por LOOM dependiendo de cada forma estrófica.
- Distinguimos entre las apariciones de las palabras que forman parte de una sinalefa. Las apariciones al principio de una sinalefa son aquellas que terminando en vocal van seguidas de otra palabra que empieza en vocal. Análogamente, las apariciones al final de una sinalefa son aquellas que empezando en vocal son precedidas por una palabra que termina en vocal.

4.1.4 Reconocimiento de instancias

Como hemos descrito en los apartados anteriores, para representar los individuos del modelo del dominio, utilizamos cierta información sobre las palabras, como el número de sílabas, la rima, si empieza o termina por vocal, o la posición del acento que se representa explícitamente, es decir, se computa y se aserta sobre los individuos que representan las palabras.

Tras los asertos iniciales, el mecanismo de reconocimiento de instancias de LOOM organiza a los individuos en torno a los conceptos a los que pertenecen, es decir, cuya definición satisfacen, permitiendo así la inferencia de conocimiento adicional. Por ejemplo, la palabra "adormecido" se representa como una instancia de `Palabra_Concept` sobre la que asertamos la siguiente información:

```
(tell (:about adormecido Palabra_Concept
   (textoPalabra "adormecido") (categoria-sintactica VLPPMS[1]) (numero-silabas-palabra 5)
(acento 4)(rima-palabra "ido")(empVocal 1)(terVocal 1)))
```

El clasificador de instancias de LOOM reconoce el individuo como instancia de los conceptos: `Cinco_Sílabas`, `Acento_Cuatro`, `Empieza_Vocal` y `Termina_Vocal`.

De manera análoga las apariciones de las palabras, los versos y las estrofas se clasifican bajo los conceptos adecuados en función de sus características. La Figura 6-13 muestra cómo se clasifica el segundo verso de la única estrofa de un poema que se reconoce como un terceto uno-tres. Las definiciones de los conceptos se pueden consultar en el Apéndice E.

Con la información inicial (parte superior de la figura) el sistema reconoce (parte inferior de la figura) que el verso es una instancia de los conceptos `Endecasílabo` y `Verso_Rimado`. Además, la única estrofa del poema se reconoce como un terceto del tipo uno-tres, es decir como instancia de `TercetoUnoTres`, ya que cada uno de sus versos se reconoce como `Endecasilabo`, y el sistema comprueba que el primer verso rima con el tercero. La relación `riman` se satisface para cada dos versos con la misma rima, es decir, con el mismo valor en el atributo `rima-verso`. Los versos primero y tercero se reconocen como `Verso_Rimado` y el segundo como `Verso_No_Rimado`, y las últimas palabras de los versos primero y tercero se reconocen como `Aparicion-Palabra-Rimada`, y la última palabra del segundo verso se reconoce como `Aparicion-Palabra-No_Rimada`.

4.1.5 Integración del modelo del dominio con CBROnto

El modelo del dominio consta de 643 conceptos, 27 relaciones y 7078 individuos (incluyendo las palabras del vocabulario). Cada relación del dominio se clasifica bajo la relación `domain-relation` de CBROnto. De la misma forma todos los conceptos del dominio se clasifican bajo el concepto `domain-concept` de CBROnto. Además hemos identificado algunos roles de conocimiento que pueden resultar interesantes para los métodos CBR:

- La relación `anterior-a-palabra` y `encabalgado-con` del dominio se clasifican bajo la relación temporal `before` de CBROnto. Esto indica un orden adicional de secuenciamiento de los elementos.
- La relación `encabalgado-con` y `riman` se clasifican bajo la relación `depends_on` de CBROnto. Estas dependencias indican que si el verso v1 está encabalgado o rima con el verso siguiente v2, las modificaciones en v1 afectan a v2 y viceversa.

[1] Verbo Léxico. Participio Presente. Masculino Singular.

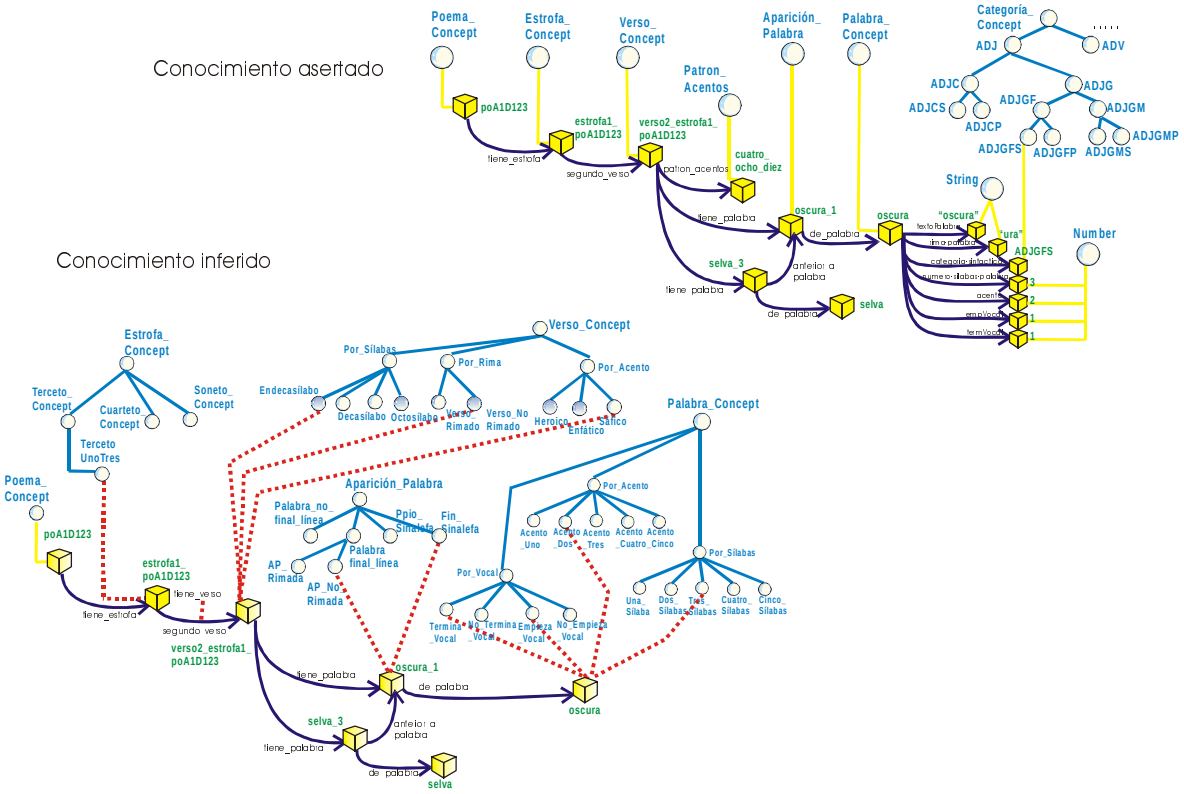


Figura 6-13. *Reconocimiento de instancias en el dominio de la poesía*

- Las relaciones `tiene-estrofa`, `tiene-verso`, `tiene-palabra`, `tiene-linea` y `de-palabra` del dominio se clasifican bajo la relación estructural `has_part` de CBROnto. Esto indica que esas relaciones son utilizadas para formar la estructura de ciertos elementos. Es importante resaltar el hecho de que esas relaciones tienen a su vez otras subrelaciones que también quedan clasificadas como estructurales. Por ejemplo, las relaciones `primer-verso`, `segundo-verso` y `tercer-verso` son subrelaciones de `tiene-verso`, las relaciones `primer-terceto`, `primer-cuarteto` son subrelaciones de `tiene-estrofa`, etc.

4.2 Casos y consultas

Una vez adquirido el modelo del dominio, el siguiente paso es utilizarlo para definir los casos que componen el corpus de poemas, que pueden estar organizados o no en distintos tipos según su estructura.

En la aplicación hemos definido un tipo llamado `tipoPoema` cuyos casos tienen descripción y solución y ambas coinciden con el poema (instancia de `Poema_Concept`) representado en cada caso. La Figura 6-14 muestra la estructura de un caso que representa un poema. Disponemos de 22 casos con distintas características y formas estróficas (1 soneto, 9 romances, 8 tercetos y 4 cuartetos). La Figura 6-15 muestra un listado de la base de casos.

También hemos distinguido distintos subtipos en función de la forma estrófica del poema `tipoTerceto`, `tipoCuarteto`, `tipoSoneto`, etc., que resultan útiles para hacer búsquedas

slgt2: TipoPoema						
has-description	slgt2:Poema_Concept					
has-solution	slgt2:Poema_Concept					
	tiene-estrofa	st1-poe-slgt2:Estrofa_Concept				
		primer-verso (tiene-verso)	l1-st1-poe-slgt2:Verso_Concept			
			primera-palabra (tiene-palabra)	no221:Aparición_Palabra		
			rima-verso	"ada"		
			numero-silabas-verso	11		
			patron-acentos	dos-seis-diez:Patrón_Acentos		
			encabalgado-con	l2-st1-poe-slgt2:Verso_Concept		
			tiene-palabra	no221: Aparición_Palabra		
			tiene-palabra	so_lo243: Aparición_Palabra		
			tiene-palabra	en413: Aparición_Palabra		
			tiene-palabra	plata485: Aparición_Palabra		
				anterior-a	o484:Aparición_Palabra	
				de-palabra	plata1:Palabra_Concept	
					textoPalabra	"plata"
					numero-silabas-palabra	2
					acento	1
					rima-palabra	"ata"
					empVocal	0
					terVocal	1
					categoria-sintactica	ADJGMS:POSTag
			tiene-palabra	o484: Aparición_Palabra		
			tiene-palabra	viola321: Aparición_Palabra		
			tiene-palabra	truncada122: Aparición_Palabra		
		segundo-verso (tiene-verso)	l2-st1-poe-slgt2: Verso_Concept			
		tercer-verso (tiene-verso)	l3-st1-poe-slgt2: Verso_Concept			

***Figura 6-14**. Estructura de un poema*

restringidas cuando el usuario está interesado en una forma estrófica concreta (atributo `casebase` del individuo que representa el contexto de aplicación).

Con la representación propuesta, la consulta del usuario es un texto distribuido en una o más líneas. Hemos representado este tipo de consultas definiendo el concepto `Texto_en_Poema`, que es subconcepto de `Query-Type` y `tipoPoema`, y que guía la definición de consultas a la construcción de un individuo con la misma estructura relacional, es decir, con los mismos atributos, que los poemas de la base de casos.

4.3 Selección y configuración de tareas y métodos

De las numerosas variaciones de interacción con el sistema propuesto el diseñador deberá decidir el comportamiento deseado para la aplicación final y configurar los métodos convenientemente. En este apartado veremos cómo llevar a cabo la configuración de tareas y métodos de varios ciclos CBR de resolución de problemas de la aplicación. El siguiente apartado describe la tarea de organización de los casos y los siguientes se encargan de la configura-

Caso: |I|A1D123
en mitad del camino de la vida
me halle_ en el medio de una selva oscura
despue_s de dar mi senda por perdida

Caso: |I|A2D456
ay cua_nto el descubrir es cosa dura
esta selva salvaje a_spera y fuerte
que en el alma renueva la amargura

Caso: |I|A3D789
amargura y pavor que es casi muerte
mas para hablar del bien alli_ encontrado
dire_ de lo dema_s que vi por suerte

Caso: |I|A4D101112
no se_ co_mo entre_ alli_ tal era el grado
de sopor que trai_a me inconsciente
cuando hube el buen camino abandonado

Caso: |I|RC1
el Campeador entonces
se dirigio_ a su posada
y en cuanto llego_ a la puerta
se la encontro_ bien cerrada

Caso: |I|SLGT2
no so_lo en plata o viola truncada
se vuelva mas tu_ y ello juntamente
en tierra en humo en polvo en sombra en nada

Caso: |I|SGDLVC1
en tanto que de rosa y de azucena
se muestra la color en vuestro gesto
y que vuestro mirar ardiente honesto
con clara luz la tempestad serena

Caso: |I|SGDLVC2
y en tanto que el cabello que en la vena
del oro se escogio_ con vuelo presto
por el hermoso cuello blanco enhiesto
el viento mueve esparce y desordena

Caso: |I| SGDLV
en tanto que de rosa y de azucena
se muestra la color en vuestro gesto
y que vuestro mirar ardiente honesto
con clara luz la tempestad serena
y en tanto que el cabello que en la vena
del oro se escogio_ con vuelo presto
por el hermoso cuello blanco enhiesto
el viento mueve esparce y desordena
coged de vuestra alegre primavera
el dulce fruto antes que el tiempo airado
cubra de nieve la hermosa cumbre
marchitara_ la rosa el viento helado
todo lo mudara_ la edad ligera
por no hacer mudanza en su costumbre

Caso: |I|SLGT1
goza cuello cabello labio y frente
antes que lo que fue en tu edad dorada
oro lilio clavel cristal luciente

Caso: |I|SLGC1
mientras por competir con tu cabello
oro brun_ido el sol relumbra en vano
mientras con menosprecio en medio el llano
mira tu blanca frente al lilio bello

Caso: |I|RC5
nueve an_os tiene la nin_a
que ante sus ojos se planta

Caso: |I|RC9
esto la nin_a le dijo
y se entro_ para la casa

Caso: |I|RC8
ya veis Cid que en nuestro mal
vos no habe_is de ganar nada
que el Creador os valga
con toda su gracia santa

Caso: |I|RC7
nadie os acoge por nada
si no es asi_ lo perdemos
lo nuestro y lo de la casa
y adema_s de lo que digo

Caso: |I|RC6
Campeador que en buen hora
cen_isteis la vuestra espada
orden del rey lo prohibe
anoche llego_ su carta

Caso: |I|RC2
mandatos del rey Alfonso
pusieron miedo en la casa
y si la puerta no rompe
no se la abrira_n por nada

Caso: |I|RC3
alli_ las gentes del Cid
con voces muy altas llaman
los de dentro que las oyen
no respondi_an palabra

Caso: |I|SLGC2
mientras a cada labio por cogello
siguen ma_s ojos que a clavel temprano
y mientras triunfa con desde_n lozano
del luciente cristal tu blanco cuello

Caso: |I|RC4
aguijo_ el Cid su caballo
y a la puerta se llegaba
del estribo saco_ el pie
y un fuerte golpe le daba

Caso: |I|SGDLVT2
marchitara_ la rosa el viento helado
todo lo mudara_ la edad ligera
por no hacer mudanza en su costumbre

Caso: |I|SGDLVT1
coged de vuestra alegre primavera
el dulce fruto antes que el tiempo airado
cubra de nieve la hermosa cumbre

Figura 6-15. *Base de casos de poemas*

ción de los métodos asociados a las tareas derivadas de la aplicación del método CBR: recuperación, reutilización, revisión y aprendizaje.

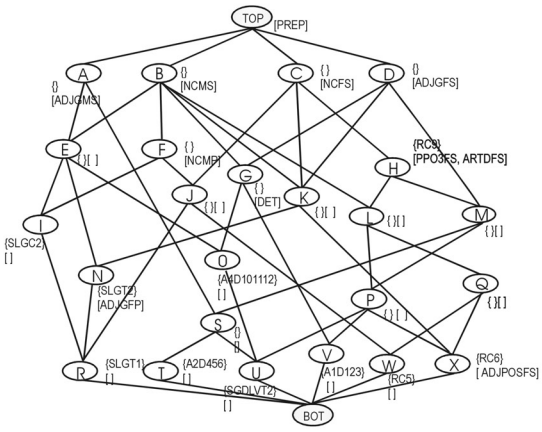
Como vimos en el Apartado 3.3.3 una de las primeras decisiones que debe tomar el diseñador de la aplicación se refiere a las tareas que se resolverán en cada ciclo CBR para posteriormente seleccionar y configurar los métodos que se usarán para resolver cada una de esas tareas en la aplicación diseñada. Los apartados siguientes ejemplifican la resolución de las tareas de organización de los casos (Apartado 4.3.1), recuperación (Apartado 4.3.2), adaptación (Apartado 4.3.3) y revisión (Apartado 4.3.4). El Apartado 4.3.5 resume las opciones de configuración que se incluirán en la aplicación, presenta ejemplos de los resultados así como las conclusiones extraídas.

4.3.1 Organización de los casos

La tarea de organización de la base de casos (`CB_Organization_Task`) es una de las tareas previas a la resolución de problemas (`CBR_Task`), cuya resolución puede llevarse a cabo por iniciativa del diseñador (al que COLIBRI muestra todas las tareas de CBROnto) o por sugerencia de los métodos que utilizan alguna estructura, por ejemplo, los métodos de recuperación basados en el retículo AFC, incluyen como prerrequisito la construcción del retículo de conceptos formales.

En este apartado computaremos una estructura para organizar los casos del ejemplo aplicando el AFC, tomando como atributos del contexto formal las categorías sintácticas de las palabras que aparecen en los poemas de los casos. En el contexto formal (G, M, I), el conjunto de objetos (G) está formado por todos los casos, el conjunto de atributos (M) por todas las categorías sintácticas (cadena de relaciones desde los individuos caso hasta los individuos que representan las categorías sintácticas), y la relación de incidencia (I) entre G y M indica cuándo un poema de un caso tiene una palabra de una cierta categoría sintáctica.

La configuración del método `iFCA_Properties_Method` consiste en especificar como requisitos de diseño las cadenas de relaciones que se utilizarán para definir el contexto formal



Reglas de dependencia :
F = {ADJGMS → PREP; NCMS → PREP; NCFS → PREP ;
ADJGFS → PREP ; NCMP → NCMS; DET → ADJGFS;
DET → NCMS; PPO3FS → ARTDFS; ARTDFS → PPO3FS;
PPO3FS → NCFS; ARTDFS → NCFS; ADJGFP → ADJGMS;
ADJGFP → NCFS; ADJGFP → ADJGFS; ADJGFP → NCMS;
ADJGFP → PREP; ADJPOSFS → PPO3FS; ADJPOSFS → ARTDFS;
ADJPOSFS → NCMS; ADJPOSFS → NCFS; ADJPOSFS → ADJGFS}

	Extensión	Intensión
TOP	TODOS	PREP
	A1D123, A4D101112, A2D456, SGDLVT2, RC6, SLGT1, SLGT2	ADJGFS, PREP
	A4D101112, A2D456, SLGC2, SGDLVT2, SLGT1, SLGT2	ADJGMS, PREP
	A1D123, A4D101112, SLGC2, SGDLVT2, RC6, RC5, SLGT1, SLGT2	NCMS, PREP
	A1D123, A2D456, SGDLVT2, RC6, RC9, RC5, SLGT1, SLGT2	NCFS, PREP
	A1D123, A2D456, SGDLVT2,RC6, RC9, RC5	PPO3FS, NCFS, ARTDFS, PREP
	A4D101112, SLGC2, SGDLVT2, SLGT1, SLGT2	ADJGMS, NCMS, PREP
	A2D456, A1D123, SGDLVT2, RC6	ADJGFS, ARTDFS, NCFS, PPO3FS, PREP
	A1D123, SGDLVT2, RC6, RC5	ARTDFS, NCFS, NCMS, PPO3FS, PREP
	A1D123,SGDLVT2,RC6	ADJGFS, ARTDFS, NCFS, PPO3FS, PREP,NCMS
	A1D123, A4D101112, SGDLVT2	ADJGFS, DET, NCMS, PREP
	SLGC2, RC5, SLGT1	NCMS, NCMP, PREP
	SLGT1, SLGT2, RC6	ADJGFS, NCFS, NCMS, PREP
	A2D456, SGDLVT2	ADJGFS, ADJGMS, ARTDFS, NCFS, PPO3FS, PREP
	SLGC2, SLGT1	ADJGMS, NCMS, NCMP, PREP
	RC5, SLGT1	NCFS, NCMS, NCMP, PREP
	A4D101112, SGDLVT2	ADJGFS, ADJGMS, DET, NCMS, PREP
	SLGT1, SLGT2	ADJGFS, ADJGFP, ADJGMS, NCFS, NCMS, PREP
	RC6, RC5	ARTDFS, NCFS, NCMS, PPO3FS, PREP
	SGDLVT2, A1D123	ADJGFS, ARTDFS, DET, NCMS, NCFS, PPO3FS, PREP
	A2D456	ADJGFS, ADJGMS, ARTDFS, NCFS, PPO3FS, PREP
	SGDLVT2	ADJGFS, ADJGMS, ARTDFS, DET, NCMS, NCFS, PPO3FS, PREP
	SLGT1	ADJGFS, ADJGFP, ADJGMS, NCFS, NCMS, NCMP PREP
	RC5	ARTDFS, NCFS, NCMS, NCMP, PPO3FS, PREP
	RC6	ADJGFS, ADJPOSFS, ARTDFS, NCFS, NCMS, PPO3FS, PREP
BOT	∅	TODOS

Figura 6-16. *Análisis Formal d e Conceptos en la base de casos de poemas*

(se guardan en el atributo `relation_path` del individuo que representa los requisitos de diseño del método):

```
(put-design-requirements iFCA_Properties_Method '((relation-path '(has-description tiene-linea tiene-palabra de-palabra categoria-sintactica))))
```

De esta forma la tabla de incidencia tiene una fila por cada caso y una columna por cada categoría sintáctica. La siguiente tabla muestra un fragmento[2] de la relación de incidencia entre los casos y las categorías sintácticas de las palabras en sus versos, y la Figura 6-16 muestra el diagrama de Hasse asociado al retículo de conceptos formales del contexto formal representado por la tabla, así como las reglas de dependencia extraídas del contexto.

	ADJ GFS	ADJ GFP	ADJ GMS	ADJ POSFS	ART DFS	DET	NCFS	NCMS	NCMP	PPO3 FS	PREP
A1D123	☑				☑	☑	☑	☑		☑	☑
A4D101112	☑		☑			☑		☑			☑
A2D456	☑		☑		☑		☑			☑	☑
SLGC2			☑					☑	☑		☑
SGDLVT2	☑		☑		☑	☑	☑	☑		☑	☑
RC6	☑			☑	☑		☑	☑		☑	☑
RC9					☑		☑			☑	☑
RC5					☑		☑	☑	☑	☑	☑
SLGT1	☑	☑	☑				☑	☑	☑		☑
SLGT2	☑	☑	☑				☑	☑			☑

La estructura construida puede resultar útil para la tarea de recuperación de casos cuya resolución se discute en el siguiente apartado.

[2] Por motivos obvios de falta de espacio.

4.3.2 Recuperación

La tarea de recuperación (`Retrieval_Task`) es la primera subtarea derivada del método de razonamiento basado en casos (`CBR_Method`) al resolver la tarea `CBR_Task`, es decir, al iniciar un ciclo de resolución de problemas. Como hemos visto en el Capítulo 5, distintas configuraciones de los métodos de recuperación de CBROnto (instancias de `Retrieval_Method`) ofrecerán comportamientos distintos, por lo que en los siguientes apartados se repasan algunas alternativas para esta aplicación.

4.3.2.1 Método de inspección de casos por complección de consultas

Una primera posibilidad que nos hemos planteado, aunque finalmente no se incluyó en la aplicación, ha sido utilizar el *método de inspección de casos* [Díaz&González00b] [Díaz&González01a], en el que se utilizan las reglas de dependencia extraídas de la aplicación del AFC (Figura 6-16) y que forman parte de los requisitos de aplicabilidad del método. Este método utiliza estas reglas para completar las consultas del usuario y llevar a cabo un proceso de recuperación exacta, en el sentido de que los casos recuperados satisfacen exactamente los requisitos de la consulta completada (considerando como requisito que se mantengan las categorías sintácticas de las palabras de la consulta). El retículo de la figura se ha construido utilizando como atributos del contexto formal (algunas de) las categorías sintácticas de las palabras que forman los poemas de los casos, este proceso supone que la consulta es interpretada como la sucesión de las categorías sintácticas de las palabras dadas. Supongamos por ejemplo que el usuario plantea una consulta como: "la casa de", cuyas categorías sintácticas son: "`ARTDFS NCFS DET`" El método aplica las reglas de dependencia para indicarle al usuario las características de una consulta *correcta*. Por ejemplo, las reglas `DET→NCMS Y DET→ADJGFS` indican que en todos los poemas de la base de casos en los que aparece un determinante aparece también un nombre común masculino singular y un adjetivo general femenino singular. Estas reglas proporcionan pautas para completar la consulta con palabras similares a las de los casos (a las que podrán sustituir durante la adaptación por tener las mismas categorías sintácticas).

Las características de corrección se determinan en base a las reglas de dependencia que se extraen de la base de casos, y no son reglas generales del dominio, aunque en este ejemplo, las reglas de dependencia representan algunas reglas de corrección gramatical para la construcción de frases en poemas en castellano[3].

Aunque podría ser interesante un estudio más detallado, en esta versión de la aplicación nos centramos en otras aproximaciones que se basan en valorar la similitud entre la consulta dada por el usuario y los poemas de la base de casos.

4.3.2.2 ¿Cómo valorar la similitud entre dos poemas?

El modelo del dominio que hemos formalizado representa conocimiento *sintáctico y métrico* sobre los poemas, los versos y las palabras, pero no incluye conocimiento *semántico* sobre ellos. Si se añadiera –por ejemplo utilizando una ontología con vocabulario (Mikrokosmos [Mahesh96]) o de conjuntos de sinónimos (Wordnet [Miller90]), lo que posponemos a una segunda versión de la aplicación– permitiría incluir valoraciones semánticas sobre las palabras. En la versión actual de la aplicación la única consideración *semántica* que podemos tener en cuenta es si en los dos textos aparecen las *mismas* palabras, o palabras con textos parecidos, y el orden relativo entre ellas.

[3] Aunque no ocurre exactamente así en el ejemplo reducido que hemos utilizado para facilitar la exposición, en el que se ha calculado el retículo sobre un subconjunto y no sobre el total de las categorías sintácticas que aparecen en los casos.

Como la consulta es un texto dividido en líneas, la similitud con los casos dependerá principalmente de la similitud entre las palabras de la consulta y las que aparecen en los poemas. Como primera opción para resolver la tarea de recuperación de casos barajamos en primer lugar el uso del *método de recuperación por cómputo de similitud numérica*, para discutir después otras opciones.

Método de recuperación por cómputo de similitud numérica

Para medir la similitud entre la consulta y los casos de forma adecuada los requisitos de conocimiento del método sugieren al diseñador la definición de medidas de similitud computacional basadas en las funciones de similitud predefinidas en CBROnto o en nuevas funciones definidas por el diseñador de la aplicación. Por tanto, utilizando la interfaz que hará uso de las funciones de la API `CreateSimilarityMeasure` y `link-similarityMeasure`, el diseñador define y asocia una medida de similitud al concepto `Poema_Concept` que indica cómo comparar dos poemas, teniendo en cuenta los atributos y la posición en la jerarquía de los individuos que los representan.

El atributo `tiene-estrofa` relaciona un poema con sus estrofas. Para comparar dos estrofas se puede optar por definir una única medida de similitud que se asocia al concepto `Estrofa_Concept` o definir medidas específicas para los distintos tipos de estrofas (subconceptos de `Estrofa_Concept`). De la misma manera definiremos medidas asociadas a los conceptos `Verso_Concept`, `Aparición_Palabra` y `Palabra_Concept` para indicar cómo profundizar en la estructura de representación de los casos (Figura 6-14), utilizando funciones globales en la componente de similitud por contenidos de las medidas de similitud, complementadas si se quiere con el uso de tipos de índice asociados a los conceptos o con funciones ponderadas para variar el peso de los distintos atributos.

Optaremos por una medida simple, común para los conceptos anteriores, combinada con la definición de índices para seleccionar qué características intervienen:

```
Función API: (CreateSimilarityMeasure common_sim
                :contents imedia :position ideep :combination imedia)
Función API: (AssociateSimilarityMeasuretoConcept Poema_Concept common_sim)
Función API: (AssociateSimilarityMeasuretoConcept Estrofa_Concept common_sim)
Función API: (AssociateSimilarityMeasuretoConcept Verso_Concept common_sim)
Función API: (AssociateSimilarityMeasuretoConcept Aparición_Palabra common_sim)
Función API: (AssociateSimilarityMeasuretoConcept Palabra_Concept common_sim)
```

Para comparar apariciones de palabras se puede asociar un tipo de índice al concepto `Aparición_Palabra` para indicar si la medida de similitud debe o no tener en cuenta el *orden* entre las palabras. Si se quiere tener en cuenta el orden el tipo de índice asociado al cómputo de similitud de dos apariciones de palabras incluirá la relación `anterior_a`. Si no se quiere tener en cuenta el tipo de índice sólo incluirá la relación `de-palabra`.

Aunque los individuos intermedios en la estructura de representación del poema, es decir, las estrofas, los versos y las apariciones de las palabras, también intervienen en la similitud, la parte más importante es la que corresponde a la similitud entre las palabras. Cuando la medida de similitud anterior (`common_sim`) se usa para comparar dos palabras (instancias de `Palabra_Concept`):

- La componente de *posición* (función profundidad) considera la posición de las palabras según cómo están clasificadas en el modelo del dominio. Recordamos que las palabras se clasifican según sus características sintácticas, como el número de sílabas, su acento, y si empiezan o no por vocal.
- La componente de *contenidos* (función media aritmética) reflejará cómo tener en cuenta los valores de las características (atributos o relaciones) que describen a las pala-

bras, es decir, el texto de la palabra, el número de sílabas, si empieza y termina por vocal, su categoría sintáctica, su rima y dónde tiene el acento.

- La componente de *combinación* (función media aritmética) indicará cómo combinar los dos resultados anteriores.

Se observa que las características que inducen clasificación sobre las palabras (como el número de sílabas o el acento) pueden intervenir tanto en la componente de posición como en la de contenidos. Sin embargo, las características que no inducen clasificación adicional, como el texto de la palabra, su categoría sintáctica o su rima, sólo intervienen en la componente de contenidos. El diseñador puede definir un tipo de índice para indicar qué relaciones utilizar, por ejemplo, sería razonable que el número de sílabas no intervenga en la componente de contenidos porque es uno de los criterios de clasificación de palabras y lo tendrá en cuenta en la componente de posición en función de cómo se clasifique la palabra. Para crear el tipo de índice y asociarlo con el concepto `Palabra_Concept` se utilizan las llamadas a las siguientes funciones de la API:

```
(Create_Index_Type pal_idx '(textoPalabra rima-palabra categoria-sintactica))
(AssociateIndextoConcept Palabra_Concept idx)
```

Para comparar el texto de las palabras (atributo `textoPalabra`) se puede utilizar la medida por similitud por defecto de CBROnto (`default_simMeasure`), que utiliza la igualdad en la componente de similitud por contenidos, o asociar al concepto `Texto` una medida cuya función de similitud por contenidos tenga en cuenta la cadena de caracteres que define la palabra. Por ejemplo, la función `maxsubcadena`, que computa un valor similitud que depende la subcadena común más larga, considera que "inconsciente" y "consciente" tienen una similitud mayor que "inconsciente" y "abandonar". Como las subcadenas comunes no siempre reflejan la similitud semántica –por ejemplo, "pecera" y "cera"– consideraremos el uso de la función de igualdad para cubrir el objetivo de recuperar los casos con más palabras en común con la consulta, por lo que la medida que se usa en los conceptos `Texto` y `Categoría_Sintáctica`, es una medida de similitud definida por defecto en CBROnto que tiene la siguiente estructura:

```
(CreateSimilarityMeasure default_simMeasure :contents iigual :position ideep :combination imedia)
```

Aunque se pueden hacer pruebas con distintas funciones de similitud hasta lograr resultados adecuados, este método tiene un inconveniente que lo hace inaplicable. El cómputo de similitud es muy ineficiente al requerir la comparación de cada palabra de la consulta con todas las de los casos. La ineficiencia inherente al método computacional se puede subsanar utilizando algún otro método para filtrar la base de casos inicial, por ejemplo alguno de los métodos basados en clasificación o en criterios de relevancia, sobre la que se selecciona usando el cómputo de similitud numérico.

Como el modelo del dominio formaliza las propiedades sintácticas de los poemas y permite su organización taxonómica en torno a ellas, configuraremos el método de r*ecuperación por reconocimiento de instancias* que es aplicable en el contexto actual y que nos permite recuperar eficientemente casos cuyas palabras en los versos estén clasificadas igual que las palabras dadas en la consulta.

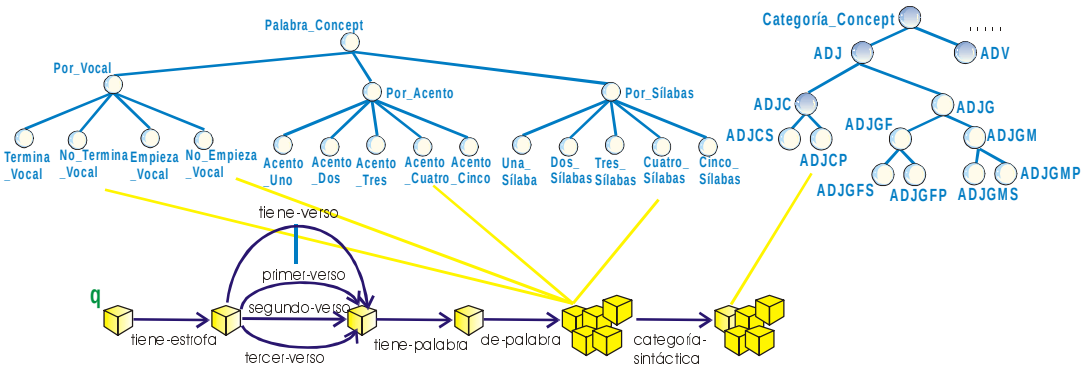


Figura 6-17. *Clasificación de instancias*

Método de recuperación por reconocimiento de instancias

La interfaz de configuración nos indicará que el método de recuperación por reconocimiento de instancias admite como requisitos de diseño la especificación de una cadena de relaciones, si lo que se quiere clasificar en el modelo del dominio no es el individuo caso (y consulta) sino una parte suya. Esto ocurre en el ejemplo actual porque nos interesa recuperar casos en función de la clasificación de sus palabras (y de las categorías sintácticas de las palabras) y no de la clasificación del individuo poema en sí mismo (que es plana). Por tanto a través de la interfaz configuraremos el método facilitando las cadenas de relaciones que unen un poema con sus palabras y con las categorías sintácticas de las palabras (ver Figura 6-17) generando las siguientes llamadas a funciones de la API:

```
(put-design-requirements 'IRetrieve_Instance_Classification_Method
 '((relation-path '(has-solution tiene-estrofa tiene-verso tiene-palabra de-palabra
                                 categoria-sintactica))
   (relation-path '(has-solution tiene-estrofa tiene-verso tiene-palabra de-palabra))))
```

Por ejemplo, el individuo que representa a la palabra "oscura" está clasificado (en el nivel más específico) como instancia de los conceptos: `No_Termina_Vocal, Acento_Dos, Empieza_Vocal y Tres_Silabas.` Esta misma clasificación la comparte con palabras como: "espalda", "amarga", "alegre", "avispa" y "abeto", entre muchas otras. El individuo que representa la categoría sintáctica de la palabra "oscura" se clasifica como `ADJGFS` (adjetivo general femenino singular) igual que las palabras "alegre" y "amarga" que recuperamos como palabras similares sintácticamente, pero no como las palabras "abeto", "avispa" o "espalda" que no son recuperadas.

Recordemos que este método de recuperación deriva dos subtareas para las que el diseñador también debe elegir métodos que las resuelvan: valoración de la similitud –`irecognize_task (AssessSim_Task)`– y selección de casos –`iselect_case (Select_Task).`

Para la tarea de valoración de la similitud, la elección del método con semántica conjuntiva `(iinstance_classification_and)` supone la recuperación de casos que tengan palabras con exactamente la misma clasificación que las palabras dadas en la consulta. La elección del método con semántica disyuntiva `(iinstance_classification_or)` supone la recuperación de casos que tengan alguna palabra clasificada exactamente igual que alguna de las palabras dadas en la consulta.

En este ejemplo sería más adecuado el uso del método con semántica conjuntiva ya que garantiza la posterior sustitución (durante la adaptación) de todas las palabras de la consulta.

Sin embargo como la base de casos no es muy grande puede ocurrir que no exista ningún caso que satisfaga exactamente las condiciones pedidas por lo que configuraremos el método para que, si no obtiene resultados, generalice la clasificación. Recordamos las opciones:

1. Realizar anotaciones (instancias del concepto `GTO_Specification`) en los conceptos que representen el modo de generalizar en una cierta zona de la taxonomía del dominio (también se utilizarán por otros métodos basados en clasificación tanto de recuperación –si hay otros ciclos– o para la búsqueda de sustitutos de adaptación). Esta opción es adecuada si el modo de generalizar es intrínseco a la taxonomía.
2. Especificar explícitamente, a través del atributo `gto_specification` de los requisitos de diseño la forma de generalizar en este método concreto. De esta forma otros métodos pueden tener configuraciones distintas, por ejemplo, un mismo concepto se podría generalizar de distinta forma al buscar casos y al buscar sustitutos.

En este ejemplo la primera opción supone anotar instancias de `GTO_Specification` en los conceptos hoja de las dos jerarquías de conceptos a las que llevan las cadenas de relaciones dadas en la configuración del método. Estas anotaciones indicarán cómo generalizar. Por ejemplo, la siguiente llamada configura los conceptos para generalizar un nivel hacia arriba y recuperar tanto las instancias directas como las no directas (lo que garantiza que siempre se recuperan instancias):

```
(create-GTO_Specification (gi :level -1 :order level_first :direct relax)
(link-GTO_Specification (gi Termina_Vocal))
(link-GTO_Specification (gi No_Termina_Vocal)) ....
```

La segunda opción consiste en configurar explícitamente la generalización mediante los requisitos de diseño:

```
(put-design-requirements 'IRetrieve_Instance_Classification_Method
 '((gto_specification '((0,true) (-1,true) (-1,false))))
```

Utilizaremos la primera opción ya que la valoración de la similitud entre la consulta y un poema se mide precisamente en función de la similitud entre las palabras que me permitirán hacer sustituciones, por lo que tiene sentido que se lleve a cabo la misma generalización de una característica durante la recuperación de casos y durante la recuperación de sustitutos.

Podemos refinar la configuración anterior del método de recuperación por reconocimiento de instancias para que tenga en cuenta también la jerarquía de relaciones. Esta nueva configuración podría considerarse de forma alternativa, o simultánea configurando dos copias del individuo canónico `IRetrieve_Instance_Classification_Method` que representa al método. La característica de la segunda configuración es la distinción explícita entre las relaciones `primer-verso, segundo-verso` y `tercer-verso` de la consulta y de los casos, en vez de utilizar la superrelación `tiene-verso` como hemos hecho en la configuración anterior del método. Estas relaciones pueden ser relajadas *un nivel* cuando no se recuperen instancias con la misma clasificación, de forma que tenemos en cuenta explícitamente el hecho de que el usuario haya colocado una cierta palabra en una línea del poema y no en otra, y que en el caso una palabra aparezca en un verso y no en otro:

```
(put-design-requirements 'IRetrieve_Instance_Classification_Method
'((relation-path '(has-solution tiene-estrofa (primer-verso -1) tiene-palabra de-palabra)))
  (relation-path '(has-solution tiene-estrofa (segundo-verso -1) tiene-palabra de-palabra)))
  (relation-path '(has-solution tiene-estrofa (tercer-verso -1) tiene-palabra de-palabra)))
  (relation-path '(has-solution tiene-estrofa (primer-verso -1) tiene-palabra de-palabra
   categoria-sintactica))
  (relation-path '(has-solution tiene-estrofa (segundo-verso -1) tiene-palabra de-palabra
```

```
   categoria-sintactica))
  (relation-path '(has-solution tiene-estrofa (tercer-verso -1) tiene-palabra de-palabra
   categoria-sintactica))
   (generalize_first 'relation))
```

El método intentará recuperar casos que tengan palabras clasificadas igual (semántica conjuntiva) y además colocadas en el mismo verso que el que se especifica en la consulta. Si no encontramos ningún caso con estas características, se relajan en primer lugar las relaciones `(generalize_first 'relation)` para indicar que queremos palabras clasificadas igual aunque aparezcan en versos distintos, y luego los conceptos para encontrar palabras clasificadas cerca. En la aplicación elegiremos esta configuración del método.

Respecto a la selección de métodos para resolver la segunda de las subtareas que genera el método de recuperación por reconocimiento de instancias, `iselect_case (Select_Task)`, el diseñador puede elegir entre los métodos cuya competencia es adecuada. El uso del método de selección por cómputo de similitud `iselect_computation_method`, que hemos configurado en el apartado anterior, permite seleccionar el poema con más palabras *iguales* a las especificadas en la consulta.

La configuración realizada para el método de recuperación de casos por reconocimiento de instancias mide la similitud entre dos poemas teniendo en cuenta únicamente la clasificación de sus palabras (y categorías sintácticas de las mismas) en una taxonomía de conceptos que reflejan aspectos sintácticos y métricos. La idea que justifica este tipo de similitud es que estamos pensando en un ciclo CBR en el que se resuelven al menos las tareas de recuperación y adaptación y en el que la recuperación tiene en cuenta la adaptación prevista para buscar los casos más fácilmente adaptables. Es decir, el objetivo de la recuperación es encontrar poemas en los que se pueda realizar el mayor número de sustituciones *adecuadas* que son las que mantienen las características sintácticas y métricas del poema.

4.3.3 Adaptación

La idea de alto nivel del comportamiento de la tarea de adaptación de casos que queremos configurar es "sustituir las palabras del poema del caso recuperado utilizando las palabras dadas en la consulta a ser posible en el mismo orden y sin perder la estructura sintáctica de los versos del poema".

Si suponemos que la consulta es una frase con sentido, el hecho de colocar sus palabras en el mismo orden en el poema hace plausible la suposición de que el poema reflejará, en cierta forma, el mensaje dado en la consulta. Para mantener la corrección sintáctica restringimos las sustituciones de las palabras del poema por otras con las mismas características sintácticas. Por tanto, el método de adaptación debe sustituir cada palabra del poema por la primera palabra de la consulta con las mismas características sintácticas si hay alguna. Cada palabra de la consulta se usa como mucho una vez.

Cuando ninguna de las palabras de la consulta es un sustituto adecuado se puede optar por dejar la palabra original del poema, aunque de esta forma se generarían poemas muy similares a los de la base de casos original sobre todo si la consulta incluye pocas palabras. En aras de la creatividad, configuraremos el método de adaptación para sustituir todas las palabras del poema, usando las palabras de la consulta cuando sea posible y palabras del resto del vocabulario cuando ninguna de las palabras dadas en la consulta sea adecuada. De esta forma el resultado será similar al poema original en la estructura sintáctica pero no en las palabras (ya que no se mantienen).

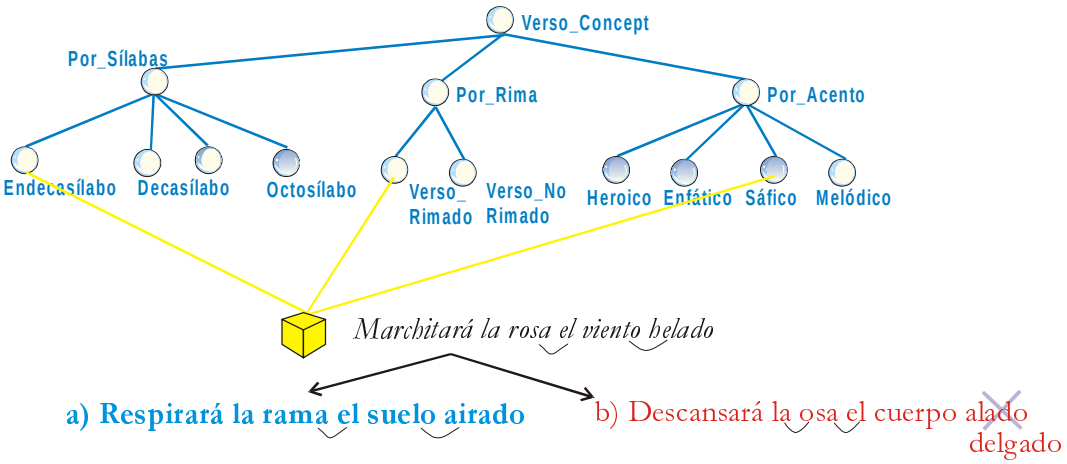


Figura 6-18. *Ejemplo de sustitución local de palabras (a) y sustitución + revisión (b)*

Respecto al mantenimiento de la estructura sintáctica y métrica de los versos del poema, las sustituciones de las palabras deberían garantizar, cuando sea posible, que se mantiene el número total de sílabas de los versos, la rima en los versos rimados y las categorías sintácticas de las palabras para mantener la coherencia de las frases (por ejemplo, en género, número y persona). Hay que tener en cuenta que estamos considerando sustituciones *locales* de palabras, es decir, las palabras se sustituirán una a una, y para cada sustitución usaremos el conocimiento (sintáctico) que tenemos sobre esa palabra en concreto pero no el conocimiento del contexto de la palabra, por ejemplo, en qué tipo de verso aparece o en qué posición del verso. En este tipo de sustituciones la única manera de mantener totalmente la estructura sintáctica de los versos del poema, es sustituir cada palabra por otra que tenga *exactamente* las mismas características sintácticas (número de sílabas, acento, rima, categoría sintáctica, empieza y/o termina por vocal), con lo que garantizamos que:

- El número de sílabas total es el mismo, ya que cada palabra se sustituye por otra con el mismo número de sílabas y que empieza y termina en vocal o no de la misma forma, con lo que no se han formado nuevas sinalefas ni deshecho ninguna de las existentes.
- La rima del poema no ha cambiado, ya que cada palabra se sustituye por otra con la misma rima (esto sólo sería necesario mantenerlo en la última palabra de cada verso).
- Las frases obtenidas tras las sustituciones son coherentes sintácticamente ya que cada palabra se sustituye por otra con la misma categoría sintáctica.

En la sustitución a) del ejemplo de la Figura 6-18, cada palabra se sustituye por otra con exactamente las mismas características sintácticas. De esta forma no se varía ninguna de las características del verso, es decir, el individuo que representa el verso (instancia de `Verso_Concept`) mantiene la misma clasificación conceptual antes y después de las sustituciones, al igual que el individuo que representa la estrofa de la que el verso formará parte.

Aunque adecuada para los objetivos de la aplicación, esta aproximación puede resultar demasiado restrictiva, es decir, una palabra debe cumplir un gran número de condiciones para ser un sustituto adecuado para otra. Esto hace que resulte difícil "colocar" las palabras dadas por el usuario en la consulta (sobre todo teniendo en cuenta de nuevo que el número

de casos no es demasiado grande por lo que las combinaciones de palabras de las que disponemos son limitadas). Sin embargo este tipo de adaptación garantiza que el resultado obtenido es *correcto* respecto a sus propiedades sintácticas y métricas, es decir, el sistema genera un poema válido de acuerdo a los criterios de corrección sintácticos establecidos por el modelo del dominio (aunque la validación semántica deberá hacerla el usuario *poeta*).

Configuraremos este comportamiento en un ciclo CBR que resuelve las tareas de recuperación y adaptación, y en el que no tiene sentido la revisión automática, ya que el sistema genera en la adaptación un poema correcto con los criterios de que dispone.

Adicionalmente, y para evitar la poca flexibilidad de la aproximación anterior, configuraremos otro ciclo CBR en el que, durante la adaptación, se permiten sustituciones que no mantengan algunas de las propiedades métricas de los versos del poema, por ejemplo el número total de sílabas o la rima, que pueden ser restauradas durante la tarea de revisión. En el ejemplo de la Figura 6-18 b) al sustituir *rosa* por *osa* no mantenemos la propiedad de finalizar en vocal y se forma una nueva sinalefa con lo que el número total de sílabas del verso varía (es incorrecto) y el verso se reclasifica como `Decasílabo` en vez de cómo `Endecasílabo`. Este error en la clasificación del verso, respecto al tipo de estrofa a la que pertenece, puede guiar al método de reparación para hacer nuevas sustituciones que restauren la clasificación original (verso endecasílabo). Por ejemplo, sustituir *alado* por *delgado* hace que se rompa una sinalefa del verso original que compensa la nueva sinalefa formada en el verso adaptado, haciendo que el verso vuelva a clasificarse como `Endecasílabo`.

El diseñador puede determinar el uso de una u otra configuración en función del tipo de usuario, del tipo de consulta o del tipo de caso. La diferencia principal entre las dos es que la segunda prima la colocación de las palabras de la consulta, es decir, las coloca en la mejor situación posible (aunque no sea óptima) y repara después los posibles errores métricos producidos. La primera configuración genera un poema con exactamente la misma estructura que el original, aunque no garantiza la colocación de todas las palabras de la consulta, sino sólo de las que satisfagan exactamente las condiciones sintácticas de alguna de las palabras del poema.

4.3.3.1 Configuración de tareas y métodos de adaptación

Recordamos que en CBROnto existe un método que resuelve la tarea de adaptación de casos descomponiéndola en dos subtareas, `Copy_Solution_Task` cuyo objetivo es hacer una copia sobre el individuo consulta de la componente de solución del caso recuperado, y `Adapt_Solution_Task` cuyo objetivo es la modificación de dicha solución. Para resolver esta última tarea existen dos métodos alternativos: el *método de adaptación especializada basada en estrategias*, que permite adaptaciones complejas con transformaciones estructurales en la solución si el diseñador describe los tipos de problemas y las estrategias que los resuelven. Y el *método de adaptación por sustitución* que resuelve la tarea de adaptación de la solución del caso recuperado sustituyendo ciertos elementos pero manteniendo la estructura de la solución del caso (es decir, no añade ni elimina elementos, sólo sustituye unos por otros).

El tipo de adaptación que queremos hacer encaja con este último método configurando la estrategia de sustitución para sustituir las palabras del poema del caso recuperado manteniendo las características sintácticas del verso al que pertenece. El método de adaptación por sustitución descompone la tarea de adaptación de la solución en tres subtareas:

- Selección de la estrategia de adaptación. El diseñador no tiene que elegir método ya que, en este método de adaptación de la solución, esta tarea está ligada a un método (`isubstitution_strategy_method`) que devuelve una estrategia genérica predefinida de sustitución de elementos que el diseñador tendrá que configurar.

- Selección de discrepancias o elementos a adaptar. Usando la estrategia de adaptación anterior se identifica la lista de acciones de transformación que se llevarán a cabo sobre la solución del caso (con esta estrategia serán sustituciones) así como los elementos involucrados en estas transformaciones.
- Modificación de la solución que se encarga de resolver la lista de discrepancias aplicando las transformaciones indicadas sobre los elementos dados.

Es importante resaltar que los elementos que serán objeto de sustitución son las palabras y no las apariciones de las palabras en los versos. Es decir, el individuo aparición seguirá siendo el mismo y así se mantiene el orden de las apariciones a través de la relación `anterior_a_palabra`. Lo que cambia es la palabra a la que se refiere esa aparición, es decir, lo que vamos a sustituir es el individuo palabra con el que se relaciona la aparición a través de la relación `de_palabra`.

Configuración de la estrategia de sustitución de elementos

El método de adaptación por sustitución de elementos utiliza una estrategia de adaptación predefinida, devuelta por el método (`isubstitution_strategy_method`) con la siguiente estructura:

```
(tellm (:about Substitution-Strategy Adaptation_Strategy (transformation-spec subst)))
(tell (:about subst Transformation (operator substitute) (applicability 'Thing)
              (search-strategy gse)))
(tell (:about gse Method-Search-Strategy
              (search-method iSearch_Instance_Classification_Method)))
```

Este método de adaptación de la solución requiere menos esfuerzo del diseñador, que el método de adaptación especializada basado en estrategias. En concreto, no es necesario que el diseñador defina los tipos de problemas de adaptación ni las estrategias de adaptación específicas para el dominio y aplicación concretos.

Sin embargo, cuando la configuración por defecto de la estrategia predefinida no es adecuada para los objetivos de la aplicación, el diseñador deberá configurar la estrategia anterior de sustitución de elementos. Esta configuración se lleva a cabo a través de los requisitos de diseño del método que devuelve la estrategia de adaptación por sustitución predefinida (`isubstitution_strategy_method`). Los requisitos de diseño del método permiten añadir alguna transformación adicional a la estrategia (que se habrá creado previamente usando alguna de las funciones de la API para definir transformaciones `create-add-transformation`, `create-delete-transformation` o `create-substitute-transformation`), y eliminar alguna de las transformaciones existentes.

Como veremos en el apartado siguiente, en este ejemplo la configuración inicial del método de búsqueda de sustitutos por reconocimiento de instancias podría resultar adecuada para mantener la estructura métrica del poema, aunque se puede mejorar su comportamiento añadiendo algunas variaciones a dicha configuración.

Selección de discrepancias (elementos a sustituir)

La siguiente tarea utiliza la estrategia de adaptación, devuelta por el método anterior, para identificar la lista de acciones de transformación que se llevarán a cabo sobre la solución del caso, así como los elementos involucrados en estas transformaciones. Como vimos en el Capítulo 5 (Apartado 4.1.3.2) COLIBRI ofrece al diseñador tres métodos que resuelven la tarea: el primero (`iuser_find_adaptation_actions_Method`) delega la resolución de la tarea al usuario final, el segundo (`ifix_adaptation_items_Method`) aplica las transformaciones

especificadas por la estrategia de adaptación a los elementos de la solución que ocupan una posición fija especificada mediante una cadena de relaciones, y el último (`isystem_find_adaptable_actions_method`) identifica los elementos sobre los que aplicar las acciones de transformación a partir de las diferencias entre el caso y la consulta.

El método (`ifix_adaptation_items_Method`) es adecuado si los candidatos a ser adaptados siempre ocupan la misma posición en la estructura de representación. Esta situación ocurre en nuestro ejemplo ya que los elementos a sustituir, las palabras, siempre ocupan la misma posición respecto a la solución del caso recuperado, es decir, al poema. El método permite configurar como requisito de diseño (atributo `toadapt`) la especificación de una lista de comprobación que determina los elementos a sustituir. El formato para especificar los individuos elegidos es una cadena `(l1, .., ln)` donde cada `li` es de la forma: `(relación restricción)` o `(relación)` y `restricción` es un concepto compatible con el rango de la relación. La cadena de relaciones especifica el camino a seguir desde el individuo a adaptar hasta los individuos a transformar –en este caso a sustituir. Además de las relaciones se pueden especificar restricciones conceptuales para las instancias del camino.

Para la aplicación elegiremos la siguiente configuración que determina que los candidatos a ser sustituidos son todas las palabras de los versos del poema recuperado:

```
(put-design-requirements ifix_adaptation_items_Method
'((toadapt'((tiene-estrofa)(tiene-verso)(tiene-palabra)(de-palabra)))))
```

Existen otras opciones, por ejemplo la siguiente configuración permitiría sustituir sólo las palabras que no estén al final del verso, lo que podría ser útil para mantener la rima del poema original al no sustituir la última palabra de cada verso:

```
(put-design-requirements ifix_adaptation_items_Method
'((toadapt '((tiene-estrofa)(tiene-verso)(tiene-palabra Palabra_No_Final_Linea)
(de-palabra)))))
```

Además, el método también admite como un *requisito paramétrico* (es decir, dado por el usuario final) una lista de individuos concretos que deben ser compatibles con la especificación de la lista de comprobación dada como requisito de diseño. Esto permite que el usuario final decida la lista concreta de palabras que quiere sustituir. Por ejemplo, la especificación:

```
(put-parameter-requirements ifix_adaptation_items_Method
    '((toadapt '(vida amargura luciente))))
```

haría que el método proponga la sustitución en el poema de las tres palabras dadas siempre que sean compatibles con la lista de comprobación de diseño, y no proponga el resto de las palabras que ocupan la posición indicada en la lista de comprobación de diseño.

En cualquiera de sus configuraciones, la salida de este método es una lista de *acciones*. Cada una de estas acciones hace referencia al elemento que estará involucrado y a la transformación que se llevará a cabo sobre él. La transformación es alguna de las que forman parte de la estrategia de adaptación, en concreto, aquella para la que el elemento cumpla su condición de aplicabilidad. En este ejemplo, como la estrategia de adaptación incluye una única transformación `subst` con aplicabilidad general (`Thing`), será siempre elegida.

Existen otras opciones además de sustituir las palabras de los poemas. Por ejemplo, sustituir versos completos del poema recuperado teniendo en cuenta sus características métricas (número de sílabas y rima). Sin embargo, no discutiremos esta opción ya que debido al tamaño reducido de la base de casos esta opción resulta en poemas muy similares a los originales.

Modificación de la solución

El método que determina los elementos a sustituir `ifix_adaptation_items_Method` devuelve una lista de acciones, que se aplican una a una, donde cada acción incluye una palabra y la transformación que realizaremos sobre ella. En este método siempre será la transformación `subst`, ya que estamos utilizando la estrategia de adaptación por sustitución:

((p_1, subst) (p_2 subst) (p_3 subst) ..... (p_N, subst))

Con la lista de acciones anterior la subtarea de modificación de la solución se encargará de aplicar las sustituciones a las palabras. El método que resuelve esta tarea la descompone en dos subtareas:

- Aplicar transformación (`iapply_transformation_task` ligada al método `apply_transformation_method` por lo que el diseñador no tiene que elegir) que aplica una a una las transformaciones de la lista de acciones resultante de la resolución de la tarea anterior.
- Revisión local de la transformación (`ilocal_revisión_task`) que se encarga de validar o rechazar cada uno de los pasos de transformación de la solución. Esta tarea es opcional por lo que simplificamos la aplicación eligiendo el método `ido_nothing_Method` para indicar que no se resolverá: `Función API: (link-task-method ilocal_revision_task ido_nothing_Method)`

Para aplicar la transformación `subst` a una palabra p_i, el método `apply_transformation_method` cuenta con la información dada en la transformación, que incluye el operador involucrado (`substitute`), la aplicabilidad (`Thing`), y la estrategia de búsqueda que hace referencia a un método de búsqueda local de sustitutos basada en el método de recuperación por reconocimiento de instancias. Este método buscará sustitutos que estén clasificados igual que el individuo que vamos a sustituir.

Como hemos visto, si este tipo de búsqueda de sustitutos no fuese adecuado, el diseñador podría variarla configurando la estrategia genérica de sustitución de elementos a través de los requisitos de diseño del método que devuelve la estrategia (`isubstitution_strategy_method`).

Otra opción, además de la posibilidad de utilizar un *método* para la búsqueda de sustitutos, es que el diseñador defina una estrategia de búsqueda en términos de las operaciones primitivas de acceso a la base de conocimiento que son necesarias para encontrar valores en la memoria, y el punto de partida –que puede ser el caso actual, la consulta o un individuo cualquiera de la base de conocimiento. La ventaja de usar una estrategia de búsqueda basada en un método, y no en operaciones de acceso a memoria, que permite valorar la similitud de la palabra y los sustitutos en cada sustitución.

En este ejemplo, usaremos el método de búsqueda de sustitutos por reconocimiento de instancias. La configuración inicial del método de búsqueda de sustitutos por reconocimiento de instancias resultaría adecuado para mantener la estructura métrica del poema sustituyendo una palabra por otra clasificada igual (semántica conjuntiva) en la taxonomía de conceptos que representan propiedades sintácticas de las palabras, en concreto, el acento, el número de sílabas y si empieza o termina por vocal, que son las propiedades que inducen la clasificación del conjunto de palabras.

Sin embargo, con la configuración inicial no estamos teniendo en cuenta la categoría sintáctica de las palabras, que no induce clasificación en el conjunto de palabras sino en los individuos que representan las categorías sintácticas. Como también hemos hecho para el método de recuperación de casos, es sencillo configurar el método de búsqueda de sustitutos

indicando el camino de relaciones que conecta una instancia consulta con las instancias que vamos a clasificar. Un detalle muy importante es que en el uso actual del método cada instancia consulta, desde el punto de vista del método de recuperación o búsqueda de sustitutos, es una *palabra* del poema que vamos a sustituir, por tanto, lo que queremos clasificar es, por un lado, la palabra en sí misma (cadena de relaciones vacía) y, por otro lado, el individuo que representa la categoría sintáctica de la palabra (cadena de relaciones `categoría-sintáctica`). Esto se especifica como:

```
(put-design-requirements ' iSearch_Instance_Classification_Method
'((relation-path '(categoría_sintáctica))) (relation-path '(nil))))
```

Con esta configuración recuperaremos como candidatas a aquellas palabras del vocabulario clasificadas de la misma forma, es decir, con el mismo número de sílabas, que empiecen y terminen en vocal de la misma forma, el acento en la misma sílaba y con la misma clasificación del individuo que representa su categoría sintáctica. Recordamos que la forma de relajar en caso de que la clasificación exacta no encuentre candidatos o éstos no sean adecuados coincide con las anotaciones realizadas en los conceptos (`GTO_Specification`) realizadas durante la configuración del método de recuperación de casos.

La configuración anterior obtiene buenos resultados en cuanto a la corrección sintáctica del poema generado. Sin embargo, utiliza palabras del vocabulario, sin tener en cuenta la consulta, que aunque se ha utilizado durante la recuperación no se tiene en cuenta como fuente de contenidos para las sustituciones llevadas a cabo en la adaptación: "sustituir las palabras del poema del caso recuperado *utilizando las palabras dadas en la consulta* a ser posible *en el mismo orden* y sin perder la estructura sintáctica de los versos del poema"

Para solucionar el problema haremos uso de un método de selección que tenga en cuenta qué palabras pertenecen a la consulta. Recordamos de nuevo que el método de recuperación por reconocimiento de instancias es un método de descomposición que deriva las subtareas de valoración de la similitud y selección de candidatos. Para la tarea de valoración de similitud se elige el método con semántica conjuntiva que recuperará palabras con exactamente la misma clasificación que las que voy a sustituir. Respecto a la tarea de selección el diseñador puede elegir entre los métodos descritos en el Capítulo 5, tanto los métodos simples, como la selección por el usuario, selección del primero o selección aleatoria, como los métodos basados en otros métodos de recuperación: selección por cómputo de similitud numérica o selección por criterios de relevancia. Además, configuraremos el método para que las dos subtareas, valoración de la similitud y selección de candidatos, se resuelvan de forma cíclica generalizando los criterios de búsqueda hasta que la salida sea no vacía.

Para la selección de sustitutos no queremos delegar la tarea de selección al usuario final sino configurar un método que se comporte adecuadamente. La selección por cómputo de similitud numérica no es adecuada porque el criterio para elegir cuál de las palabras utilizar como sustituto *no depende de la similitud entre la palabra y su sustituta*, que es lo que computaría este método para decidir qué candidato es mejor, sino de las palabras que aparecen en la consulta. Es decir, del conjunto de candidatos queremos elegir palabras que aparezcan en la consulta si es posible en el mismo verso que la que voy a sustituir. Sin embargo, el método de selección por criterios de relevancia puede resultar adecuado para especificar distintas fuentes de sustitutos ya que seleccionará de los elementos candidatos (que recibe como requisito de secuencia) los elementos que satisfacen un cierto criterio (requisito de diseño del método). Por ejemplo, el criterio `query-words` selecciona las palabras de la consulta:

```
(defrelation query-words
   :is (:satisfies (?p ?q)
          (:and (Query-Type ?q) (Palabra_Concept ?p)
```

```
(:for-some (?e ?l ?ap)
(:and
 (tiene-estrofa ?q ?e) (Estrofa_Concept ?e)
 (tiene-verso ?q ?l) (Verso_Concept ?l)
 (tiene-palabra ?l ?ap) (Aparición-Palabra ?ap)
 (de-palabra ?ap ?p))))))
```

Para configurar el comportamiento del método de selección definiremos varios criterios para que se prueben en orden, en primer lugar buscaremos sustitutos en el primer verso de la consulta (el criterio es igual que `query-words` sustituyendo `(tiene-verso ?q ?l)` por `(primer-verso ?q ?l)`, luego en el segundo y así sucesivamente. Por último, si ninguna palabra de la consulta se selecciona elegiremos una de las del vocabulario general (criterio que recupera las instancias de `Palabra_Concept`):

```
(put-design-requirements 'iselect_RelCriteria_Method
((relcri_list '(query-first-verse-words query-second-verse-words query-third-verse-words query-fourth-verse-words vocabulary-words))
(used_criteria order))
```

Para la selección posterior a la aplicación del criterio (se resuelve `select_case`) puede hacerse aleatoriamente, ya que sólo incluye palabras válidas sintácticamente, o puede hacerla el usuario, para intentar mantener algún tipo de coherencia semántica.

Ejemplo de adaptación

Supongamos que para la consulta: " descansará la flor" "en la noche helada" "y no cambiar su color" se recupera el siguiente caso que vamos a adaptar:

Caso: |I|SGDLVT2_ADAPTED

marchitara_ la rosa el viento helado

todo lo mudara_ la edad ligera

por no hacer mudanza en su costumbre

El método que determina los elementos a sustituir devuelve una lista de acciones que indica que sustituiremos todas las palabras del poema:

```
((marchitara1 subst) (la1 subst) (rosa1 subst) ...)
```

Para sustituir cada palabra se recuperan como candidatas aquellas palabras del vocabulario con la misma clasificación, tanto del individuo palabra como de su categoría sintáctica. Los candidatos para sustituir a la palabra "marchitará" serían: "marchitará", "respirará", "razonará", "dialogará", "descansará", "terminará", "discutirá", "maquillará" y "conectará".

El criterio de selección de candidatos elige aquellas palabras que pertenecen a la consulta, línea por línea, para determinar que la palabra "descansará" (que aparece en la primera línea de la consulta) es el mejor sustituto y llevará a cabo la sustitución correspondiente.

Para sustituir la palabra "rosa" existe una larga lista de candidatos con las mismas características sintácticas: "cama", "gata", "piedra", "hora", "lengua", "silla", "burra", "noche", "loba", "copa", "vida", "pata", ..., y un largo etcétera, de los que el criterio de selección de candidatos elige la palabra "noche" que pertenece a la segunda línea de la consulta. La palabra "flor" de la primera línea no se elige porque no tiene las mismas características sintácticas que la palabra a sustituir. En concreto, no comparte la característica de terminar por vocal.

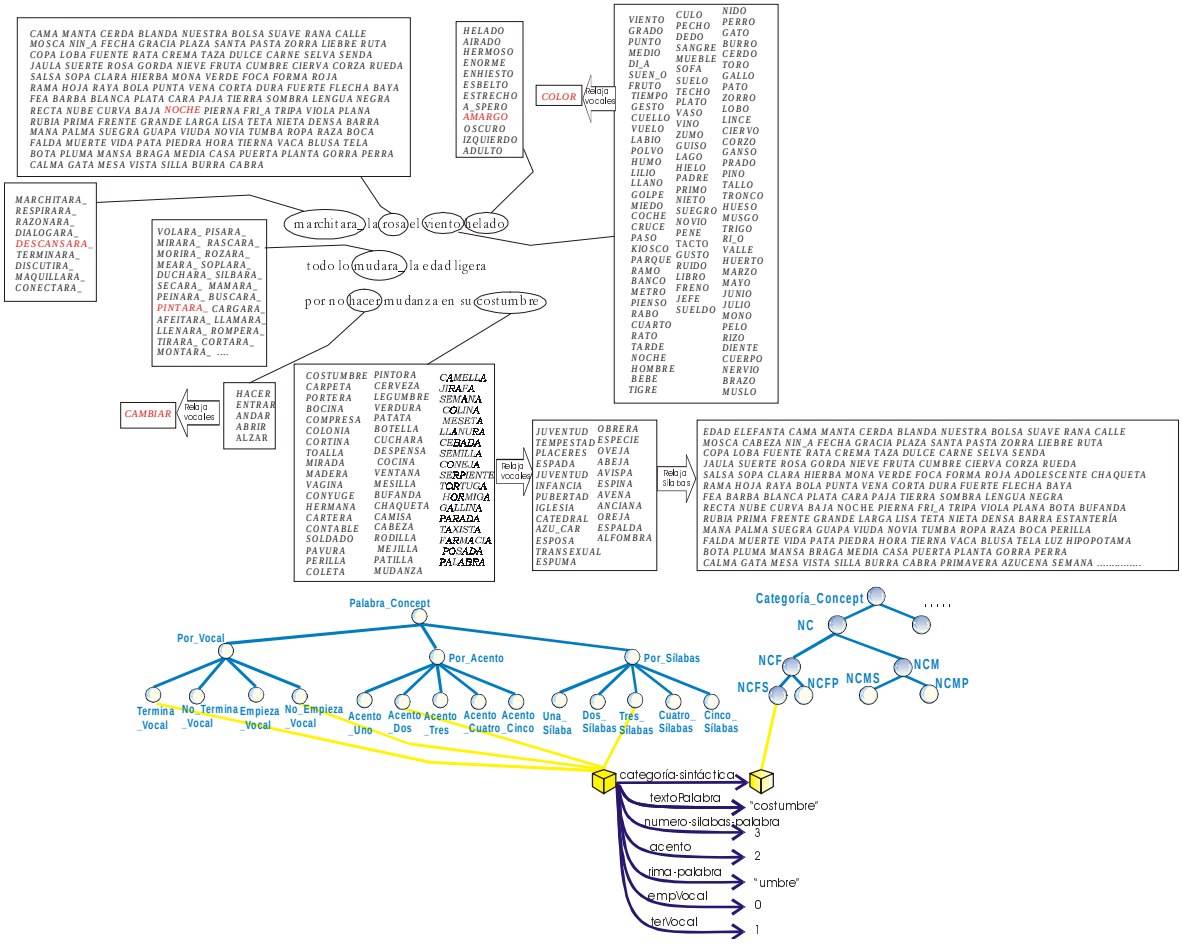


Figura 6-19. *Ejemplo de adaptación*

Para la palabra "el" no existen candidatos en la consulta por lo que se eligen candidatos del resto del vocabulario, aunque en el caso de los artículos los únicos sustitutos adecuados son ellos mismos.

Para sustituir a la palabra "viento" el sistema encuentra las palabras clasificadas de la misma forma por ejemplo "grado", "punto", "medio", "labio", "polvo", "nieto", "primo" o "tiempo", entre muchas otras, de las que el criterio de selección no encuentra ninguna que pertenezca a la consulta. Por esta razón todas ellas (que forman parte del vocabulario) son candidatas. Como hemos comentado, existe una subtarea de selección posterior a la aplicación del criterio (`select_case`). Esta selección afecta sólo a las palabras que satisfacen el criterio y dentro de ellas podemos elegir aleatoriamente o permitir que el usuario elija una de ellas. En la aplicación final configuraremos las dos opciones.

Con el método de selección aleatoria una de estas palabras siempre es elegida, pero si la selección la lleva a cabo el usuario puede que ninguna le parezca adecuada. En este caso el método de recuperación por reconocimiento de instancias, procede a resolver sus subtareas –valoración de la similitud y selección de candidatos– hasta generar un resultado no vacío. De esta forma se generaliza la clasificación como indican las anotaciones

`GTO_Specification`, es decir, un nivel hacia arriba, lo que permite que el usuario escoja palabras que no satisfacen exactamente las características de la palabra que se sustituye.

Como vimos en la Figura 6-18 b) la generalización de las características sintácticas puede suponer la violación de alguna de las características del verso o de la estrofa a los que pertenece la palabra sustituida. Como describimos posteriormente, hemos configurado la tarea de reparación para que pueda arreglar fallos en el número de sílabas y en la rima de la palabra. La rima no se está teniendo en cuenta, ya que no interviene en la clasificación, y el número de sílabas puede variar al sustituir una palabra por otra de distinta longitud o con distintas propiedades de empezar y terminar por vocal. Como son las características que sabremos reparar, son éstas las únicas que se generalizan y no ascenderemos a los superconceptos del acento ni de la categoría sintáctica (que se configuran con nivel 0 en su especificación GTO). Entre las opciones para la palabra "viento" que resultan de relajar la propiedad de terminar por vocal se encuentra la palabra "color" que pertenece a la consulta y que es elegida como sustituto.

La Figura 6-19 muestra el esquema del proceso de sustitución de palabras para obtener como resultado el poema siguiente que necesitará reparación del número de sílabas en el primer y tercer verso.

Caso: |I|SGDLVT2_ADAPTED

descansara_ la noche el color amargo

todo lo pintara_ la flor helada

por no cambiar mirada en su primavera

4.3.3.2 Ciclo CBR con adaptación por sustituciones estrictas

En el ejemplo anterior hemos visto que la configuración del método de búsqueda de sustitutos basada en el reconocimiento de instancias intenta sustituir cada palabra por otra con el mismo número de sílabas, misma posición del acento, mismas características de empezar y terminar por vocal y misma categoría sintáctica. Aunque, como hemos visto en el apartado anterior, la selección final por parte del usuario puede hacer que se relajen algunas características que estropeen la longitud del verso, en general, el verso obtenido será correcto salvo en la rima, que no participa en la clasificación del individuo.

Como hemos decidido que uno de los ciclos que configuraremos resolverá la tarea de reparación, podemos optar por dejar que sea esta tarea la que valore la rima y la repare. Sin embargo, en el ciclo CBR en el que sólo se resuelven las tareas de recuperación y adaptación, deberíamos evitar que se relajen las condiciones de clasificación, modificando las anotaciones GTO o utilizando selección aleatoria que siempre elige una palabra del conjunto inicial de opciones correctas. Con esto mantendríamos todas las características, incluyendo el número de sílabas del verso, salvo la rima que requiere añadir un criterio de selección adicional para que cada palabra se sustituya por otra con la misma rima.

Además, la aproximación de sustituir palabras por otras con las mismas características sintácticas y la misma rima limita mucho las posibilidades y, además, esto sólo es necesario en la última palabra de los versos y no en todas. Es más, sólo haría falta en la última palabra del verso si el verso es rimado. Es decir, en el segundo verso de un terceto uno tres no es necesario que la última palabra mantenga la rima, pero sí en el primero y en el tercero. Esto significa que los criterios para discernir entre los candidatos pueden variar de unas palabras a otras, en concreto dependiendo de la *aparición* concreta de la palabra.

Con estas consideraciones no basta con utilizar únicamente los atributos que describen las palabras (instancias de `Palabra_Concept`) sino que hay que tener en cuenta la *aparición*

concreta de la palabra (la instancia de `Aparición_Palabra`) que describe el contexto en el que se usa la palabra.

Para ello habría que especificar que los elementos que vamos a sustituir son las apariciones y no las palabras (aunque en la transformación añadiremos el atributo `tosubstitute` con la cadena `de-palabra` para indicar que lo que realmente sustituiremos es la palabra de la aparición), lo que resulta útil para disponer de información adicional sobre la apariciones concretas de las palabras para valorar la similitud. La clasificación de los individuos que representan las apariciones de las palabras indica características relativas a la posición, por ejemplo, si la aparición forma parte o no de una sinalefa, si está al final de un verso o no y si es una palabra rimada, es decir, que aparezca al final de un verso rimado. Por tanto, para buscar los sustitutos adecuados no utilizaríamos la clasificación de las aparición de las palabra sino la de las palabras que la componen (indicando las cadenas de relaciones que configuran el método adecuadamente teniendo en cuenta que el punto de partida es la aparición de la palabra, es decir, utilizando las cadenas (`de-palabra categoria-sintactica) y (de-palabra).`

De esta forma, después de la valoración de similitud llevada a cabo por reconocimiento de instancias, se puede elegir como método de selección de candidatos el método basado en criterios de relevancia descrito, que como vimos también permite realizar una selección posterior usando cómputo de similitud. Es decir, de los candidatos que cumplen las características sintácticas de la palabra a sustituir, se seleccionan aquellas que pertenecen a la consulta –cuando haya alguna y si no se mantienen todas– y para cada una de ellas se computa la similitud numérica con la aparición de palabra que vamos a sustituir.

Este método resulta adecuado porque permite utilizar medidas de similitud y tipos de índice diferentes, que se asocian a los conceptos adecuados (subconceptos de `Aparición_Palabra`) dependiendo de las características de cada aparición concreta. Por ejemplo, en las apariciones rimadas se tendrá en cuenta la rima y en las que forman parte de sinalefa se tiene en cuenta la característica de empezar y terminar por vocal. El método elige qué medida de similitud dependiendo de la clasificación de la aparición de palabra considerada.

No detallaremos más esta configuración (que resulta algo más complicada) que obtiene poemas en los que cada palabra se sustituye por otra de las mismas características, incluyendo la rima y las vocales al principio y final de palabra sólo cuando sea necesario, y el resto de las características: número de sílabas, acento y categoría sintáctica en todas las apariciones. En su lugar nos centramos en una aproximación mixta que permite que durante la adaptación, según describimos en el apartado anterior, se *estropeen* ciertas características sintácticas del poema, como la rima y el número de sílabas, que serán evaluadas y reparadas durante la resolución de la tarea de revisión.

4.3.4 Revisión

En vez de transformar la solución únicamente durante la adaptación de casos, hemos visto que resultaría más flexible y versátil la configuración de un ciclo CBR en el que las transformaciones se repartieran entre las tareas de adaptación y revisión. Además esta configuración nos permite ejemplificar la resolución de la tarea de revisión de casos. En concreto, hemos visto que la búsqueda de sustitutos basada en el reconocimiento de instancias nos permite sustituir una palabra por otra con el mismo número de sílabas, mismas características de empezar y terminar por vocal y misma categoría sintáctica. La selección de palabras nos permite elegir las palabras de la consulta si alguna satisface los criterios y usar palabras del resto del vocabulario si ninguna de la consulta es adecuada. Además, la configuración del método indicará cómo relajar en caso de que el usuario no seleccione ninguna palabra de las preseleccionadas por la aplicación.

Si no se relaja ninguno de los conceptos bajo los que se clasifica la palabra que vamos a sustituir, podemos asegurar que el verso obtenido es correcto sintácticamente salvo en la rima que no participa en la clasificación del individuo. Al relajar en la jerarquía de conceptos puede ocurrir que perdamos otras características del verso, en concreto el número de sílabas, por lo que estos dos serán los tipos de fallo que se repararán durante la tarea de revisión.

Hemos visto que la representación explícita del modelo del dominio permite expresar las características que describen las reglas de la poesía formal mediante definiciones terminológicas. Esto permite también que se pueda comprobar la corrección de una solución (un poema) comprobando si satisface estas características. Los conceptos del dominio bajo los que se clasifica el caso recuperado son utilizados como la descripción declarativa de las propiedades que deben ser mantenidas por el caso adaptado después de las transformaciones llevadas a cabo durante la adaptación. Es decir, los objetivos que debe satisfacer un caso correcto se corresponden directamente con los conceptos a los que pertenece el caso recuperado. La representación explícita y razonamiento con las descripciones de estos conceptos permiten valorar la corrección de un poema generado por el sistema para su posterior reparación.

Como se describió en el Capítulo 4, existe un único método de revisión `iRevise_Method`, que descompone la tarea de revisión en dos subtareas: revisión automática y revisión manual. Nos centraremos en la tarea de revisión automática que tiene asociado un método que la descompone en dos subtareas: evaluación y reparación. El método de evaluación automática se basa en clasificar el caso adaptado y comparar los conceptos bajo los que está clasificado con la clasificación del caso recuperado.

Para la aplicación que nos ocupa, la evaluación identifica los tipos de fallo en función de la clasificación de los individuos que describen el poema, tanto el propio poema (instancia de `Poema_Concept`), como sus componentes, es decir, sus estrofas, los versos que forman estas estrofas, y las palabras que forman los versos.

Cada tipo de fallo se representa como un subconcepto del concepto `FAIL_TYPE` de CBROnto, con una definición adecuada para que el individuo involucrado en el problema se reconozca como una instancia de ese concepto cuando ocurre dicho problema. Además cada tipo de fallo es anotado con una estrategia de reparación para ese problema específico. La función de la API `create-fail-type` se encarga de crear el concepto tipo de fallo. La función `associate-repair-strategy` asocia una estrategia de reparación con un tipo de fallo. Una estrategia de reparación `st` es una instancia del concepto de CBROnto `repair_strategy` y se une con el tipo de fallo `ft` a través de la relación de CBROnto `has_repair_strategy`. La aplicación de una estrategia de reparación corregirá el fallo y el individuo involucrado vuelva a clasificarse como estaba y deje de pertenecer al tipo de fallo.

La función que define los tipos de fallo permite tres opciones, que varían dependiendo de cómo reconoce a sus instancias el concepto tipo de fallo: cuando dejan de pertenecer a un concepto dado (`out`), cuando se reconocen como instancias de un concepto dado (`in`), o cuando dejan de pertenecer a un concepto y se reconocen como instancia de otro. Además, cuando haya varios conceptos involucrados se puede indicar su uso conjuntivo (`and`) o disyuntivo (`or`), por ejemplo, reconocer un fallo cuando un individuo deje de pertenecer a C1 **y** a C2 **y** a C3, y se reconozca como instancia de C1 **o** de C2 **o** de C3:

```
(create-fail-type FT '((out (and C1 C2 C3)) (in (or C1 C2 C3)))))
```

Para esta aplicación hemos identificado dos tipos de fallo:

- `Fallo_Rima`: significa que un poema debería rimar y no lo hace. Los conceptos que participan en este fallo son los conceptos que representan las formas estróficas rimadas, es decir, `TercetoUnoTres` y `Cuarteto`. También están relacionados los con-

ceptos `Verso_Rimado`, que reconoce como instancias suyas a los versos que forman parte de las estrofas anteriores y que deben rimar, y `Aparición_Palabra_Rimada` que reconoce a las últimas palabras de los versos rimados. Con esta representación, cuando un individuo deja de pertenecer a cualquiera de los conceptos anteriores se identifica un fallo de rima en el poema adaptado. Trabajaremos con las estrofas, de forma que un individuo estrofa (instancia de `Estrofa_Concept`) se clasifica bajo el concepto que representa el tipo de fallo cuando deja de pertenecer a los conceptos `TercetoUnoTres` o `Cuarteto.` El tipo de fallo se define a través de la siguiente llamada a la función de la API :

```
(create-fail-type Fallo_Rima '(out (or TercetoUnoTres Cuarteto)))
```

- `Fallo_Numero_Silabas`: significa que el número de sílabas de alguno de los versos del poema no es adecuado. Los conceptos involucrados son `Octosílabo` y `Endecasílabo`, que son los que reconocen a los versos válidos para las formas estróficas representadas en la base de casos. Al definir este tipo de fallo deberíamos tener en cuenta que es importante si el número de sílabas falla por exceso o por defecto, ya que esto influirá en la estrategia de reparación que se utilice. Por tanto, definiremos los dos tipos de fallo siguientes[4]:

```
(create-fail-type Faltan_Silabas '(or   ((out endecasilabo) (in decasilabo))
  ((out endecasilabo) (in eneasilabo))((out endecasilabo) (in octosilabo))
  ((out endecasilabo) (in heptasilabo)) ((out octosilabo) (in heptasilabo))
  ((out octosilabo) (in sexsasilabo))   ((out octosilabo) (in pentasilabo))))
(create-fail-type Sobran_Sílabas '(or   ((out endecasilabo) (in dodecasilabo))
  ((out endecasilabo) (in masdeoncesilabo))
  ((out octosilabo) (in eneasilabo))    ((out octosilabo) (in decasilabo))
  ((out octosilabo) (in endecasilabo))))
```

Una vez identificados los tipos de fallo comienza la subtarea de reparación en la que los problemas encontrados sirven de índices para recuperar las estrategias de reparación adecuadas. El método de reparación automática deriva dos subtareas –encontrar estrategia y aplicar estrategia– que se repiten de forma cíclica hasta que no haya mas fallos o para alguno de los fallos existentes no se pueda aplicar la estrategia.

Aunque este método es genérico en cuanto a su estructura, su funcionamiento depende directamente de la definición de tipos de fallo y estrategias de reparación adecuadas para esta aplicación concreta. Como describimos en el Capítulo 5, para encontrar la estrategia el método busca en la jerarquía de tipos de fallo el concepto más específico que tenga una estrategia asociada y que tenga alguna instancia directa y aplica la estrategia asociada a los individuos clasificados bajo el concepto que representa el tipo de fallo. Al igual que las estrategias de adaptación, una estrategia de reparación puede definirse como una función con código Lisp o haciendo referencia a alguno de los métodos de CBROnto configurado adecuadamente.

Para esta aplicación hemos identificado las siguientes estrategias de reparación que se asocian a los conceptos que representan los tipos de fallo:

- `Repair_Rima`: se asocia al concepto `Fallo_Rima` y se aplica cuando alguna estrofa del poema se haya clasificado bajo dicho concepto.
- `Repair_Faltan_Silabas`: se asocia al concepto `Faltan_Sílabas` y se aplica cuando un verso del poema se haya clasificado bajo este concepto. Esta estrategia consiste en sustituir la palabra más corta del verso que no pertenezca a la consulta por otra

[4] también se podría definir un concepto adicional en el modelo del dominio que represente a todos los versos de menos de once sílabas para no tener que distinguirlos uno por uno.

con una sílaba más manteniendo el resto de las características. Si hay más de una palabra con la longitud más corta se escoge una al azar.

- `Repair_Sobran_Silabas`:
  Se asocia al concepto `Sobran_Sílabas` y se aplica cuando un verso del poema se haya clasificado bajo este concepto. La reparación consiste en sustituir la palabra más larga del verso que no pertenezca a la consulta por otra con una sílaba menos manteniendo el resto de las características.

Las estrategias se definen a través de la función de la API `create-repair-strategy`, que crea una instancia de `repair-strategy` que puede estar ligada con una función Lisp (por si se quieren hacer cosas muy específicas) o con un método de la biblioteca configurado (a través de sus requisitos) adecuadamente. El comportamiento del método que aplica una estrategia sobre un individuo es genérico, por ejemplo, `aplica-estrategia (repair-faltan-sílabas,V1)`

1. Si la estrategia tiene código Lisp (relación `lisp-function`) aplicar la función pasándole como parámetro la instancia `V1`. `Apply (f, V1)`
2. Si la estrategia tiene asociado un método de la biblioteca (relación `CBROnto-method`) el diseñador no escribe código Lisp sino que configura un método de la biblioteca de CBROnto a través de sus requisitos. La función hace la llamada al método usando el individuo clasificado como fallo para establecer la entrada (requisitos de secuencia o paramétricos).
3. Si la estrategia tiene ambos, primero se ejecuta la función y luego el método. Esta combinación permite llevar a cabo la configuración del método en la función Lisp, en vez de hacer una configuración fija, lo que permite variar la configuración entre distintas reparaciones en función de las características de la entrada.

En la estrategia **`Repair_Faltan_Silabas`** queremos *sustituir* la palabra más corta del verso por otra con una sílaba más, lo que parece que podremos solucionar aprovechando el método de la biblioteca de CBROnto de adaptación por sustitución de palabras:

1. Hacemos una copia del método elegido –*método de adaptación por sustitución*– que será la que se configure llamando a la función de la API `copy-method`, que se asocia a la estrategia.
   ```
   (create-repair-strategy Repair_Faltan_Silabas (:method (copy-method
   iAdapt_by_Substitution_Method)))
   ```
2. Este método deriva subtareas asociadas a otros métodos que hay que configurar:

   (a). Establecemos los requisitos del método que encuentra los elementos a adaptar para que lo que sustituya sea la palabra más corta. Elegiremos el método que encuentra los individuos que ocupan una posición fija (copia del individuo canónico `ifix_adaptation_items_Method`) y recordamos que el formato para especificar los individuos a sustituir es una cadena (l1, .., ln) donde cada li es de la forma: (relación restricción) o (relación) y restricción es un concepto compatible con el rango de la relación. La cadena de relaciones especifica el camino a seguir desde el individuo a adaptar hasta los individuos a sustituir. El método permite también especificar como requisito de diseño (atributo `filter`) una función que ordena la lista de elementos obtenidos según algún criterio. Para la situación que nos ocupa no se puede expresar una restricción conceptual que indique "la palabra más corta del verso), por lo que utilizamos una cadena de relaciones que nos lleve a todas las palabras que no sean de la consulta, para mantener las palabras dadas por el usuario, y las ordenamos de menor a mayor longitud usando una función externa.

```
(put-design-requirements 'ifix_adaptation_items_Method '((toadapt '((tiene-
palabra not-query-component) (de-palabra))))(filter 'mas_corta))
```

(b). Respecto a la estrategia de adaptación predefinida, devuelta por el método `isubstitution_strategy_method` vamos a modificar la estrategia de búsqueda para que en vez de utilizar la recuperación de sustitutos por clasificación (como hemos hecho para la adaptación) utilice el método basado en criterios de relevancia que permite hacer recuperaciones más exactas puesto que usa un criterio concreto. Configuramos el método `iSearch_Relevance_ Criteria_Method` con un criterio de relevancia que busca sustitutos con una sílaba más que la palabra actual y el resto de las características se mantengan.

La estrategia `Repair_Rima` se asocia al concepto `falla_rima` bajo el cuál se habrán clasificado aquellos individuos estrofa en los que ha fallado la rima. La estrategia se encarga de sustituir las palabras al final de los versos rimados de la estrofa. Sin embargo, existe un problema debido a que el criterio para sustituir una palabra no depende de las características de esta palabra sino de otra palabra de otro verso, con la que debe rimar, es decir tener el mismo valor en el atributo rima. El método basado en criterios de relevancia podría resultar adecuado si pudiéramos definir un criterio adecuado, lo que no es posible porque el criterio debería ser variable.

Una opción es definir una estrategia de reparación cuya especificación operacional se define por una función y por un método, de forma que la función Lisp configura el criterio de relevancia que usará el método. Sin embargo, ni siquiera el criterio es fijo para cada individuo fallido. Es decir, en el mismo individuo (estrofa) los criterios pueden ser distintos para distintas palabras, por ejemplo en un cuarteto pueden fallar las dos estrofas. Por esto hemos decidido definir la estrategia mediante una función Lisp que utiliza métodos de la biblioteca pero asume el control en las llamadas.

```
(create-repair-strategy Repair_Rima (:function configura_criterio_rima))
```

La función tiene acceso a la estrofa completa, que es el individuo clasificado como fallido, así que puede definir un criterio adecuado para buscar sustitutos manteniendo la rima del poema. Por ejemplo, en el siguiente poema adaptado falla la rima porque durante la adaptación se ha sustituido la última palabra del primer verso *vida* por *nada*, que pertenece a la consulta del usuario:

Caso: |I|A1D123_ADAPTED

en mitad del camino de la *nada*

me halle_ en el medio de una selva oscura

despue_s de dar mi senda por perdida

Durante la resolución de la tarea de reparación se detecta un fallo de rima y se aplica la estrategia de reparación `Repair_Rima.` La lista de palabras a sustituir estará formada por una única palabra: '(perdida), que debo sustituir por otra de las mismas características sintácticas y que rime igual que "nada".

La función define el criterio de similitud `similar_con_rima` y hace la llamada siguiente para encontrar palabras que rimen con "nada" y con las mismas características que "perdida", encontrando palabras como "rizada" o "casada" de las que elige una aleatoriamente (o puede mostrárselas al usuario para que elija una de ellas). Como la rima es más difícil de mantener intentaremos incluir el menor número de restricciones en la búsqueda, por lo que obviamos las características relativas al número de sílabas que podrán ser reparadas en la estrategia asociada a los fallos de longitud de los versos:

```
(retrieve (?x ?y) (:and (similar_con_rima ?x ?y)))
```

```
(eval `(retrieve (?y) (:and (similar_con_rima ,(fi nada) ?y))))
(defrelation similar_con_rima :is (:satisfies
                                    (?p1 ?p2)
                                    (:and (Palabra_Concept ?p1)
                                          (Palabra_Concept ?p2)
                                          (:same-as  (acento ?p1) (acento ?p2))
                                          (:same-as  (rima-palabra ?p1) (rima-palabra ?p2))
                                          (:for-some (?x ?y ?z)
                                                  (:and
                                                    (proper-subrelations categoria_concept ?y)
                                                      (categoria-sintactica ?p1 ?x)
                                                      (categoria-sintactica ?p2 ?z)
                                                      (instancia_directa ?x ?y)
                                                      (instancia_directa ?z ?y) )) )))
```

4.3.5 La aplicación final

En los apartados anteriores hemos descrito algunas características de la aplicación diseñada que resumimos en este apartado. Hemos definido un tipo de casos representado por el concepto `Tipo_Poema`, y varios tipos de casos adicionales (subconceptos del anterior) que representan distintos tipos de poemas según sus formas estróficas y que resultan útiles para hacer recuperaciones restringidas a un tipo de poemas (estableciendo el atributo `casebase` del individuo que representa el contexto de aplicación). El usuario final puede definir consultas de un único tipo representado por el concepto `Texto_en_Poema` que se corresponde con un texto dividido o no en varias líneas con el mensaje que pretende transmitir el poema que se construya, y que tiene la misma estructura que los casos.

Respecto a las tareas y los métodos hemos descrito dos ciclos CBR:

- Ciclo 1: recuperación y adaptación sin revisión. Las sustituciones en la adaptación son estrictas en el sentido de que sustituyo una palabra del poema por otra de la consulta exactamente de las mismas características, incluyendo la rima en las palabras finales del verso. No hace falta revisión porque sabemos que el poema no se ha estropeado. Las sustituciones se hacen manteniendo totalmente las características sintácticas de corrección del poema.
- Ciclo 2: recuperación, adaptación y revisión. Las sustituciones pueden estropear la rima y el número de sílabas que serán reparadas durante la revisión. Si la reparación falla el poema se devuelve al usuario.

Para cada uno de los dos ciclos anteriores hemos configurado dos variaciones que se refieren al método de selección final del candidato. El usuario puede elegir entre selección aleatoria o selección manual, tanto para la selección de casos como para la selección de palabras sustitutas.

En ambos ciclos, si el usuario plantea una consulta vacía o una consulta con una sola palabra muy genérica, por ejemplo "el", que está en todos los poemas, el sistema le ofrece todos los casos de la base de casos elegida (tipo de caso) para que elija uno manualmente y sobre él realiza las sustituciones dando a elegir al usuario qué palabra de las candidatas quiere utilizar. Este tipo de interacción corresponde con un proceso en el que el sistema hace las comprobaciones sintácticas de corrección y ayuda al usuario a crear un poema correcto.

El siguiente apartado muestra algunos ejemplos de los resultados generados por la aplicación diseñada donde se puede observar que los criterios de corrección son sintácticos y sólo se puede hablar de corrección en la semántica de los poemas obtenidos cuando es el usuario el que selecciona las palabras sustitutas.

4.3.5.1 Ejemplos de poemas generados

Distinguimos entre los resultados obtenidos con las distintas posibilidades de configuración de los ciclos CBR descritos.

Ciclo 1. Recuperación + Adaptación (estricta) sin revisión

Con selección aleatoria de candidatos

Sin consulta

Caso: |I|A2D456_ADAPTED
ay cua_nto el descansar es barra dura
esta nieve profunda a_spera y blanca
que en el oído renueva la abertura

Poema original:
ay cua_nto el descubrir es cosa dura
esta selva salvaje a_spera y fuerte
que en el alma renueva la amargura

Caso: |I|SGDLVT2_ADAPTED
razonara_ la jaula el burro esbelto
todo lo buscara_ la osa soltera
por no entrar colina en tu pa_jara

Poema original:
marchitara_ la rosa el viento helado
todo lo mudara_ la edad ligera
por no hacer mudanza en su costumbre

Caso: |I|A4D101112_ADAPTED
no vi_ co_mo alze_ alla_ tal era el vado
de lunes que trai_a me amarillo
cuando hube el gris aspecto apartado

Poema original:
no se_ co_mo entre_ alli_ tal era el grado
de sopor que trai_a me inconsciente
cuando hube el buen camino abandonado

Con consulta:

"tumba de piedra sellada en la oscura sombra de la noche"

Caso: |I|SLGT2_ADAPTED
no so_lo en *tumba* o *piedra sellada*
se viese mas tu_ y ello fijamente
en *sombra* en suelo en *noche* en media en nada

Poema original:
no so_lo en plata o viola truncada
se vuelva mas tu_ y ello juntamente
en tierra en humo en polvo en sombra en nada

"verde valle parque noche novia junio"

Caso: |I|RC4_ADAPTED
afeito_ el Cid su lagarto
y a la *noche* se secaba
del aceite cambio_ el pie
y un *verde parque* le daba

Poema original:
aguijo_ el Cid su caballo
y a la puerta se llegaba
del estribo saco_ el pie
y un fuerte golpe le daba

Caso: |I|RC9_ADAPTED
esto la *novia* le cupo
y se alzo_ cambia la *noche*

Poema original:
esto la nin_a le dijo
y se entro_ para la casa

Con selección manual de candidatos (obligatorio seleccionar alguna de las opciones)

Sin consulta

Caso: |I|SGDLVT2_ADAPTED
descansara_ la tumba el tacto amargo
todo lo cambiara_ la edad madura
por no alzar mirada en tu destino

Poema original:
marchitara_ la rosa el viento helado
todo lo mudara_ la edad ligera
por no hacer mudanza en su costumbre

Caso: |I|SGDLVT1_ADAPTED
cambiad de vuestra hermosa catarata
el verde parque antes que el lago oscuro
pinte de suerte la herida senda

Poema original:
coged de vuestra alegre primavera
el dulce fruto antes que el tiempo airado
cubra de nieve la hermosa cumbre

Caso: |I|A3D789_ADAPTED
almohada y jabo_n que es casi suerte
mas para saltar del bien alli_ afeitado
sabre_ de lo dema_s que vi por fuerte

Poema original:
amargura y pavor que es casi muerte
mas para hablar del bien alli_ encontrado
dire_ de lo dema_s que vi por suerte

Caso: |I|SLGC2_ADAPTED
mientras a cada cuerpo con cogello
peinan ma_s arcos que a delfi_n hermano
y mientras vuelve con calor verano
del delgado metal su duro sello

Poema original:
mientras a cada labio por cogello
siguen ma_s ojos que a clavel temprano
y mientras triunfa con desde_n lozano
del luciente cristal tu blanco cuello

Con consulta:

"verde valle parque noche novia junio"

Caso: |I|SLGT2_ADAPTED
no so_lo en clara o *noche* sellada
se viese mas tu_ y ello fijamente
en *noche* en *parque* en *junio* en cumbre en nada

Poema original:
no so_lo en plata o viola truncada
se vuelva mas tu_ y ello juntamente
en tierra en humo en polvo en sombra en nada

Caso: |I|SLGT1_ADAPTED
palpa temor pasado *valle* y *novia*
antes que lo que fue en su edad ruidosa
ancho *junio* color final pálido

Poema original:
goza cuello cabello labio y frente
antes que lo que fue en tu edad dorada
oro lilio clavel cristal luciente

Ciclo 2. Recuperación + Adaptación + Revisión

Con selección manual de candidatos

Marcamos en negrita las palabras para las que se han generalizado sus características. En el caso de la rima no es que se generalice sino que no se tiene en cuenta.

Sin consulta

Caso: |I|SLGC1_ADAPTED

cuando por descubrir con tu **recuerdo**

rizo dorado el color **pinta** en vano

tanto con pensamiento en **ancho** el **camino**

sopla su limpia fuente al nido **airado**

Poema original:

mientras por competir con tu cabello

oro brun_ido el sol relumbra en vano

mientras con menosprecio en medio el llano

mira tu blanca frente al lilio bello

Revisión: durante la evaluación se identifica un fallo de rima en la estrofa que se clasifica bajo el concepto `Fallo_Rima` por haber dejado de pertenecer al concepto `Cuarteto` al que pertenecía. Además, el individuo que representa al tercer verso se clasifica bajo el concepto `Sobran_Sílabas` por haber dejado de pertenecer al concepto `Endecasílabo` para pasar a ser instancia del concepto `Dodecasílabo`.

Durante la reparación se aplican las estrategias correspondientes. Recordamos que se aplica en primer lugar la estrategia que repara los fallos de rima ya que puede variar el número de sílabas del verso. Aplicamos la estrategia de reparación asociada al concepto `Fallo_Rima`. Como no tenemos una consulta la lista de palabras a sustituir estará formada por las palabras que aparecen al final de los cuatro versos, ya que todos deben rimar: (recuerdo vano camino airado).

Las sustituciones comienzan por la primera palabra de la lista "recuerdo" que debe rimar igual que "airado". El sistema busca palabras con la misma categoría sintáctica que "recuerdo", es decir, NCMS (nombre común masculino singular) y la rima de "airado", para encontrar por ejemplo palabras como "grado", "subordinado", "apartado", "soldado", "abogado", "recado", "pasado", "tejado", "pescado" o "asado". Elegiremos "pasado" que resulta más adecuada respecto a la semántica del verso. Con esta sustitución la palabra "airado" deja de ser candidata a ser sustituida.

La siguiente palabra de la lista es "vano" que debe ser sustituida por una palabra de la misma categoría sintáctica que "vano" y que rime con "camino". Como no existe ninguna palabra del vocabulario que satisfaga estas restricciones el sistema prueba con la siguiente palabra de la lista, antes de producir un fallo de reparación. Para sustituir la palabra "camino" por alguna que rime con "vano" el sistema encuentra varias candidatas como "cirujano", "hermano", "plano", "llano", "verano"o "gusano". Elegiremos como sustituta la palabra "verano" lo que produce que la estrofa vuelva a rimar y sea reconocida de nuevo como una instancia de `Cuarteto`:

Caso: |I|SLGC1_ADAPTED

cuando por descubrir con tu pasado

rizo dorado el color pinta en vano

tanto con pensamiento en ancho el verano

sopla su limpia fuente al nido airado

El siguiente paso es la aplicación de la estrategia de reparación del número de sílabas del tercer verso. La estrategia `Repair_Sobran_Sílabas` intentará sustituir la palabra más larga del verso, en este caso, "pensamiento" por una palabra con las mismas características aunque con una sílaba menos, por ejemplo, "recuerdo", "camino", "pasado", "futuro", "presente", "domingo", "febrero", "verano", "momento", "gorila", "camello" o "flequillo". De las que elegiremos la palabra "futuro". El resultado es el siguiente poema que es correcto respecto a las propiedades métricas y sintácticas del modelo del dominio:

Caso: |I|SLGC1_ADAPTED
cuando por descubrir con tu pasado
rizo dorado el color pinta en vano
tanto con *futuro* en ancho el verano
sopla su limpia fuente al nido airado

4.3.5.2 Conclusiones, trabajo relacionado y extensiones de la aplicación de generación de poesía

En este apartado se ha presentado una versión básica de una aplicación que genera poemas en castellano [Díaz *et al.* 02]. El objetivo ha sido ilustrar con un ejemplo algo más complejo los métodos ofrecidos por CBROnto y el proceso de configuración llevado a cabo durante el diseño de una aplicación. El objetivo de la versión actual de la aplicación ha sido el de estudiar las posibilidades que ofrece el uso de un modelo de conocimiento del dominio frente al uso de otras técnicas utilizadas en las versiones previas de la aplicación.

La aplicación diseñada ofrece dos modos de uso. Un uso poco participativo en el que el sistema genera el poema usando las palabras dadas en la consulta, cuando pueden ser colocadas, y palabras del vocabulario seleccionadas aleatoriamente en otro caso. Y un uso más participativo en el que el sistema trabaja como un ayudante para el usuario poeta, ofreciéndole distintas opciones para sustituir las palabras, y reparando las características estropeadas en las sustituciones.

En la aplicación que hemos presentado no se utilizan símbolos de puntuación, no se respetan las mayúsculas al principio de línea, sólo se considera la rima consonante de las palabras y se representan las tildes mediante guiones bajos. La mejora de estas características forma parte del trabajo futuro que se incluirá en las versiones posteriores de la aplicación.

De los experimentos realizados podemos concluir que los métodos de CBROnto son adecuados para la versión básica presentada aunque existen algunas limitaciones que se deben principalmente a la carencia de conocimiento gramatical y sobre la semántica de las palabras. Aunque existen incoherencias semánticas en algunos de los poemas presentados, las restricciones formales en cuanto a métrica y cadencia prosódica son correctas. Además de estudiar si los métodos de CBROnto son adecuados para llevar a cabo la funcionalidad básica de las versiones previas de la aplicación, COLIBRI nos ha proporcionado el punto de partida para obtener nueva funcionalidad, y ha abierto varias líneas de trabajo futuro, principalmente respecto a la incorporación de propiedades semánticas del vocabulario.

El sistema WASP [Gervás00] (*Wishful Automatic Spanish Poet*) es un sistema basado en reglas con razonamiento hacia delante. El sistema trabaja con un algoritmo *generate and test* donde el proceso de generación tiene dos grados de libertad: selección del conjunto de categorías sintácticas a utilizar para el verso siguiente, y selección de la palabra a utilizar para sustituir la siguiente categoría del patrón de trabajo, que se llevan a cabo de manera aleatoria. El modelo computacional ampliado de ASPID [Gervás01a] mejora el comportamiento del sistema WASP a cambio de utilizar una cantidad importante de conocimiento en forma de vocabulario, corpus de versos etiquetados y definición de formas estróficas. El uso de un corpus de versos etiquetados, que se utilizan como referencia para la construcción de la solución, es una aplicación de la tecnología CBR y es el punto de partida para nuestra versión de la aplicación utilizando COLIBRI y CBROnto. En ASPID se utilizaba un tipo de CBR muy simplificado, sólo de recuperación de versos etiquetados de una cierta forma y una formalización algorítmica de los aspectos de la métrica tradicional que se tienen en cuenta en la generación de poesías. El modelo usado en ASPID evolucionó a ASPERA [Gervás01b] que incluye una

interfaz amigable para el usuario utilizando un sistema experto basado en reglas que a través de un proceso interactivo "inteligente" establece los parámetros de configuración del sistema de generación de poesía. El uso de COLIBRI y CBROnto y la aproximación KI-CBR al problema evita el uso de mecanismos de planificación y sustitución utilizando reglas y mecanismos algorítmicos. La alternativa que proponemos es la representación explícita de conocimiento terminológico sobre el dominio, y el uso de los métodos genéricos de CBROnto que sacan partido del mismo.

Una extensión a la funcionalidad básica incluirá conocimiento semántico basado en Mikrokosmos [Mahesh96], para completar el vocabulario con información semántica que (con variaciones mínimas de configuración) será utilizado por los métodos de CBROnto.

Por ahora los poemas se obtienen por sustitución local de palabras de forma que la corrección sintáctica del resultado se garantiza al sustituir una palabra por otra que en ese contexto no varía las características, o las varía para que el sistema pueda repararlas en una fase posterior. Aunque hemos definido los criterios de evaluación sintáctica, una fuente de incertidumbre a la hora de evaluar semánticamente los poemas obtenidos reside en la dificultad de definir un método riguroso y repetible de evaluación de los poemas obtenidos. Para el sistema WASP se presentó una selección de los poemas generados a un conjunto de evaluadores. Sin embargo la valoración mediante métodos de este tipo es siempre altamente subjetiva por cuanto que varía mucho de un individuo a otro y que las valoraciones de un mismo individuo pueden presentar variaciones entre el primer poema leído y los siguientes, que siempre se van comparando inconscientemente con los ya vistos.

Además, otro aspecto en el que extenderemos la versión actual y que esperamos nos proporcione mejores resultados (en cuanto a creatividad) es la relajación de la categoría sintáctica de la palabra sustituida. En este sentido la taxonomía de etiquetas sintácticas, que en la versión actual se utiliza superficialmente, resultaría muy adecuada para relajar una categoría como ADJFS a la categoría padre ADJF de forma que podríamos sustituir la palabra por otra que, aunque también es adjetivo, varía el número (plural), o relajando un nivel adicional, por otra que varía en género (masculino). Por ejemplo, para sustituir la palabra "guapa", además de otros adjetivos de exactamente la misma categoría –como "baja", "mansa", "fea", "plana", "gorda", "negra", "clara", etc.– el sistema puede proponer otros adjetivos –como "claro", "gordos", "guapos", "feo", "feos", etc. En la versión actual de la aplicación, en la que se sustituye una palabra por otra de exactamente la misma categoría sintáctica, podemos asegurar que se mantiene el contexto de la palabra sustituida. Sin embargo, el problema obvio de la extensión propuesta es que los cambios afectan a la corrección del verso generado. La validación de la corrección se puede llevar a cabo usando una función específica del dominio, lo que nos llevaría al comportamiento actual: elegir una palabra del mismo grupo sintáctico.

Para solucionar el problema tendríamos que utilizar el sistema de propagación de dependencias de COLIBRI, en el que si existe una dependencia entre A y B, entonces si el elemento A se sustituye por otro que está clasificado igual no pasa nada; pero si el elemento A se sustituye por otro A' que está clasificado de otra forma, entonces hay que sustituir el elemento dependiente B por otro B', de forma que la clasificación de B' se relaciona con la clasificación de B, de la misma forma que la clasificación de A' se relaciona con la de A.

El método general supone indicar cómo afectan ciertas relaciones a la sustitución y en particular lo hemos aplicado a la relacióne *es un*. Las dependencias entre elementos se deben marcar durante la fase de diseño, a nivel terminológico –como la relación `encabalgado-con` o `riman`– que se hereda para todas las instancias o a nivel de individuos concretos donde el

diseñador puede marcar una dependencia explícita, por ejemplo, entre ciertas apariciones de las palabras pero no en otras.

La aproximación usando dependencias supondrá resolver ciertos inconvenientes. Por ejemplo, resulta muy laborioso tener que marcar explícitamente las dependencias entre los individuos concretos. Por eso intentaremos realizarlas a nivel terminológico incluyendo en el modelo del dominio la representación explícita de las reglas de concordancia entre las categorías sintácticas. Además, COLIBRI no gestiona dependencias circulares y en la aplicación estas dependencias existen, por ejemplo en "la casa" hay un ciclo de dependencias ya que los cambios en cualquiera de las palabras afectan a la otra. Esto se podría solucionar considerando grupos de palabras dependientes en los que cada palabra del grupo depende de las otras. Queremos extender la aplicación incluyendo sustituciones de grupos de palabras en vez de sustituciones locales de palabras. De esta forma podemos sustituir un grupo por otro de las mismas características, en concreto, con el mismo número de sílabas y categoría.

Por último, respecto al aprendizaje, en la versión actual de la aplicación únicamente hemos considerado la inclusión de nuevos poemas que servirán como casos para posteriores interacciones. Creemos que la consideración de conocimiento semántico dará pie a la utilización de los métodos de aprendizaje de CBROnto y el aprendizaje de estrategias para relajar en la taxonomía de conocimiento semántico, aprendiendo de cómo se haya hecho otras veces (sobre todo aprendiendo de las sustituciones manuales del usuario) o que permitan aprender nuevas dependencias.

5. Resumen y conclusiones del capítulo

En este capítulo hemos presentado el modelo de desarrollo de aplicaciones KI-CBR propuesto por el sistema COLIBRI y que se basa en reutilizar, por un lado la terminología, tareas y métodos CBR de CBROnto, y por otro lado, ontologías con conocimiento del dominio extraídas de una biblioteca de ontologías. Hemos descrito la arquitectura de COLIBRI y las fases de diseño de una aplicación CBR, que hemos ejemplificado con dos aplicaciones. Una primera aplicación sencilla para gestionar la compra/venta de coches, en la que hemos visto algunos aspectos interesantes de nuestro sistema, como la reutilización de ontologías del dominio de una biblioteca de ontologías y parte de la interfaz de COLIBRI, que en el resto del capítulo se ha obviado. En la segunda parte del capítulo hemos diseñado una aplicación de generación de poesías en castellano [Díaz *et al.* 02]. Esta aplicación ha servido fundamentalmente para ilustrar la resolución de las tareas de adaptación y revisión, que en los ejemplos anteriores, por ser más simples, no habían sido objeto de atención.

El modelo terminológico del dominio ofrece una ventaja en esta aplicación, que permite utilizar una aproximación distinta de la que usan en la práctica la mayoría de los sistemas CBR para afrontar la evaluación de la corrección de una solución. En concreto los sistemas CBR utilizan una descripción implícita de la tarea que resuelven que está incrustada en el motor CBR, en vez de una descripción explícita basada en el modelo del dominio como proponemos en nuestra aproximación.

En nuestra aproximación al CBR el conocimiento de la biblioteca de casos se complementa con otras fuentes de conocimiento, principalmente con conocimiento sobre la terminología del dominio. Aunque el coste de adquisición de este conocimiento puede ser una traba inicial, la reutilización de ontologías puede aliviarlo. En nuestra arquitectura se asume un coste de adquisición inicial a cambio de disponer de métodos CBR más inteligentes que aprovechan este conocimiento y obtienen mejores resultados que con un razonamiento basado únicamente en los casos.

Capítulo 7
CONCLUSIONES Y TRABAJO FUTURO

En esta memoria de tesis hemos descrito una aproximación al desarrollo de sistemas KI-CBR que integra el uso de ontologías, mecanismos de razonamiento de las DLs, concretamente los del sistema LOOM, y métodos genéricos de resolución de problemas. En particular hemos descrito CBROnto, nuestra ontología con conocimiento de CBR, que es el núcleo del sistema COLIBRI, un entorno de desarrollo de aplicaciones CBR.

El primer apartado de este capítulo sintetiza las ideas principales y conclusiones de esta tesis. Los Apartados 2 y 3 resumen, respectivamente, los puntos que consideramos como principales aportaciones de nuestra propuesta así como los inconvenientes de la misma, algunos de los cuales intentaremos solventar con nuestras líneas de trabajo futuro.

1. Conclusiones

Son varias las conclusiones que hemos extraído durante la realización de este trabajo de tesis. Un primer aspecto compartido con otros trabajos se refiere al hecho de que los sistemas CBR pueden realizar mejores razonamientos si completan el conocimiento de los casos con un depósito de conocimiento general acerca del dominio. Además, según se refleja en el modelo de diseño de aplicaciones con COLIBRI y en los ejemplos incluidos en esta tesis, las ontologías son componentes reutilizables idóneos para el desarrollo de sistemas CBR, ya que ofrecen conocimiento previamente adquirido, conceptualizado y formalizado, lo que reduce considerablemente el coste de adquisición de conocimiento y proporciona beneficios en cuanto a su fiabilidad y consistencia.

Por otro lado, el uso de una biblioteca de métodos CBR que razonen y saquen partido del conocimiento terminológico presente en las ontologías, junto con el uso de terminología CBR estándar que facilite la integración del conocimiento del dominio con el conocimiento requerido por los métodos, facilita el proceso de diseño de aplicaciones CBR complejas.

El objetivo fundamental con el que comenzó el trabajo llevado a cabo en esta tesis fue la descripción de un marco general con buenas propiedades que utilice la lógica descriptiva para definir los algoritmos involucrados en el razonamiento basado en casos. La conceptualización y formalización de CBROnto satisface este objetivo.

La necesidad de una terminología unificadora sobre CBR nos llevó a definir CBROnto, una ontología que captura términos semánticamente importantes para los métodos CBR y proporciona vocabulario típicamente involucrado en los sistemas de CBR, tanto primitivas de representación como conocimiento de las tareas y métodos CBR. La terminología de CBROnto es relativa a CBR en general, es independiente del dominio de aplicación, y se usa como un puente entre el conocimiento del dominio y los métodos CBR genéricos.

En la practica la mayoría de los sistemas CBR utilizan conocimiento adicional sobre el dominio que está fuertemente ligado al motor CBR. Desde una postura metodológica, es mejor considerarlos separadamente y de distintos orígenes. El modelo de diseño de aplicaciones con COLIBRI se basa en la disgregación entre los componentes de un sistema CBR que pueden provenir de distintas fuentes: los casos, el conocimiento del dominio y los PSMs que los utilizan. En concreto, los PSMs y la terminología CBR que se usa como elemento integrador provienen de CBROnto y el conocimiento del dominio proviene de la reutilización de ontologías de una biblioteca o de la conceptualización y formalización de una base de conocimiento específica para la aplicación.

La reutilización de los PSMs y del conocimiento general del dominio puede permitir que el esfuerzo de desarrollo de un nuevo sistema CBR se centre en adquirir el conocimiento especializado del sistema y en incorporar mecanismos específicos de la tarea a resolver.

La elección de un lenguaje como LOOM, basado en la lógica descriptiva, para formalizar CBROnto se debe a las buenas propiedades de esta tecnología y su adecuación para el CBR que constatan los numerosos trabajos en el área [Kamp96][Napoli *et al.* 96] [Napoli&Lieber96][Napoli *et al.* 97][Lieber&Napoli98][Salotti&Ventos98][Díaz&González00a] [Díaz&González01c] [Gómez-Albarrán00]. Las DLs se caracterizan por su expresividad y su semántica claramente definida, unifican y dan fundamento lógico a los tradicionales sistemas de marcos, redes semánticas y representaciones orientadas a objetos.

En COLIBRI las ontologías ayudan a crear estructuras de conocimiento complejas que dan soporte a los procesos de razonamiento con casos. El conocimiento sobre el dominio supone conocimiento contextual semántico para los casos y permite interpretarlos, comprenderlos y medir la similitud entre ellos, o interpretar las diferencias entre las descripciones y usarlas durante la adaptación o revisión de los casos. Sin embargo, el conocimiento presente en una ontología del dominio es genérico, reutilizable y no se ha concebido específicamente para su uso en CBR. Para que los métodos genéricos de la biblioteca de CBROnto puedan aprovechar el conocimiento del dominio adquirido por reutilización de ontologías es necesario algún tipo de integración terminológica. En COLIBRI este mecanismo se basa en clasificar los términos del dominio con respecto a los términos de CBROnto, permitiendo que los métodos CBR sean independientes del dominio haciendo referencia únicamente a los términos de CBROnto, que estarán respectivamente enlazados a la terminología del dominio mediante clasificación de términos. De esta forma, la teoría específica del dominio es intercambiable y el mismo conocimiento puede jugar distintos papeles en distintos contextos de resolución de problemas usando una integración terminológica distinta.

Para concluir este apartado, resumimos las conclusiones que hemos extraído de la adquisición de conocimiento basado en la reutilización de ontologías y del uso de un sistema de DLs para representar el conocimiento de los sistemas KI-CBR.

1.1 Adquisición de conocimiento basado en la reutilización de ontologías para sistemas KI-CBR

En nuestro grupo de investigación se han desarrollado algunos sistemas CBR [González-Calero97][Gómez-Albarrán00] en los que se han formalizado bases de conocimiento *ad hoc* con el consiguiente coste de adquisición asociado. Con el espíritu de promover la reutilización de conocimiento en esta tesis se propone basar la adquisición de conocimiento para un sistema CBR en la reutilización de ontologías de una biblioteca.

De acuerdo con nuestra experiencia en el uso de COLIBRI y según ciertos estudios sobre la ganancia de productividad al reutilizar conocimiento previo en forma de ontologías [Cohen *et al.* 99]) podemos concluir los siguientes puntos:

- El conocimiento existente en las ontologías del dominio es adecuado para su uso en los sistemas KI-CBR.
- Suponiendo que existen ontologías del dominio que se pueden reutilizar, el coste total de adquisición es mucho menor que el coste de desarrollar un modelo terminológico del dominio. Esta afirmación se mantiene incluso contemplando el incremento del coste total derivado del coste de integración de conocimiento (que puede resultar difícil para el diseñador) en el caso de que las características de la aplicación requirieran la reutilización de varias ontologías.
- La adecuación de nuestra propuesta de reutilización de conocimiento del dominio depende directamente de la existencia de una biblioteca que proporcione una o varias ontologías sobre el dominio de aplicación.
- Aunque nuestro modelo se puede generalizar a otros sistemas de DLs, la implementación actual de COLIBRI en LOOM requiere la disponibilidad de una biblioteca de ontologías formalizadas en dicho lenguaje o el uso de traductores adecuados.
- Aparte de la reducción en el coste de adquisición de conocimiento, la reutilización de ontologías presenta numerosas ventajas añadidas para su uso en un sistema CBR, derivadas del hecho de disponer de una terminología estándar y probada, que representa el conocimiento consensuado de una comunidad.
- Cuando existen casos codificados previamente al desarrollo de la aplicación, el uso de ontologías del dominio puede suponer un esfuerzo de integración adicional entre la terminología de la ontología y la terminología utilizada en los casos. Sin embargo, cuando no existen casos previos, la ontología proporciona terminología estándar que puede guiar la definición de casos más completos e intercambiables.

1.2 Uso de las DLs en sistemas KI-CBR

Las DLs presentan una serie de ventajas como formalismo de representación del conocimiento ontológico en general que hemos señalado en el Capítulo 3. Resumimos a continuación las ventajas de utilizar sistemas de DLs para representar el conocimiento de tipo terminológico que se incluye en los sistemas KI-CBR:

- La clasificación automática de conceptos y el reconocimiento de instancias reducen el esfuerzo que conlleva la organización de la información existente y facilitan la incorporación posterior de otra información adicional que queda integrada con el conocimiento previo.

- Estos mecanismos de razonamiento ayudan en el diseño de ontologías, y de bases de conocimiento en general. Por ejemplo, chequean la consistencia en ontologías diseñadas por varios autores. Los mecanismos de razonamiento de las DLs también ayudan en el uso de ontologías. En particular ayudan a determinar la consistencia de un conjunto de hechos respecto a la ontología, ayudan a determinar las instancias de las clases de la ontología y ayudan en los procesos de acceso al conocimiento, donde se puede construir un concepto (o individuo) que represente una consulta y clasificarlo en la taxonomía.
- El mecanismo de detección de incoherencias ayuda en el proceso de adquisición del conocimiento inicial y en posteriores incorporaciones, permitiendo verificar la consistencia de la representación, lo que facilita también la integración de nuevo conocimiento.
- La sintaxis de las DLs es muy cercana al lenguaje natural. Este hecho junto con la definición precisa –mediante un lenguaje formal– de los términos usados en la representación –conceptos y relaciones– y su organización en una jerarquía de abstracción, ayudan al usuario en la comprensión de la información. De este modo, un usuario que no esté familiarizado con el dominio puede obtener una visión comprensiva de sus elementos y de sus interrelaciones, tal y como han sido identificados por el experto que ha construido el sistema.
- Permite el refinamiento o evolución de las descripciones. En contraste con los almacenes de datos estándar, una base de conocimiento en DLs permite que el usuario mantenga una visión parcial del universo en la que se adquiere información de forma incremental.
- La superior expresividad de las DLs (por ejemplo, restricciones entre relaciones, definición de relaciones inversas) permite plasmar aspectos importantes de la semántica de la información que no es posible reflejar usando los lenguajes de especificación de bases de datos.

Además de las ventajas como formalismo para representar conocimiento terminológico del dominio, las DLs también presentan ventajas derivadas de su uso para la representación de casos, de índices y como soporte para los métodos CBR. Éstas se plasman en que:

- Su capacidad para construir descripciones estructuradas de los casos proporciona una forma expresiva de representarlos.
- Aplicadas a la representación de los índices, permiten definir estos con una semántica formal que posibilita su organización en una jerarquía de abstracción que se construye a partir de sus definiciones y que facilita la comprensión de los atributos utilizados en la indexación de los casos.
- Los mecanismos de clasificación de conceptos y de reconocimiento de instancias de las DLs, junto con la capacidad de detectar incoherencias, facilitan la construcción y extensión automáticas del esquema de indexación.
- Los mecanismos de razonamiento de las DLs también sirven como soporte a los distintos procesos CBR. En particular el proceso de recuperación puede basarse en uno de estos mecanismos:
  - El intérprete de consultas. Permite plantear una consulta –en un lenguaje basado en la lógica de predicados– para recuperar información de la base de conocimiento.

  - El clasificador de conceptos. Permite construir una descripción de un concepto, a partir de las restricciones establecidas en la consulta. Esta descripción se clasifica en la taxonomía de conceptos para obtener los individuos que son instancias suyas.
  - El reconocedor de instancias. Permite construir un individuo genérico sobre el que se realizan asertos correspondientes a las restricciones especificadas en la consulta. Gracias al razonamiento hacia adelante, se pueden inferir hechos adicionales que enriquecen la consulta original (compleción de instancias). Seguidamente se reconoce el individuo y se recuperan los restantes individuos que son instancias del concepto más específico del que el individuo consulta es instancia.
- La posibilidad de realizar consultas incompletas, y de llevar a cabo una recuperación aproximada, es una ventaja y una característica común de todo sistema de CBR.

2. Aportaciones

La aportación global de esta tesis es la propuesta de una ontología con conocimiento sobre CBR que es el núcleo de la arquitectura del sistema COLIBRI, un entorno de desarrollo de aplicaciones CBR intensivas en conocimiento. Las aportaciones específicas de nuestro trabajo son:

- Conceptualización y formalización de CBROnto, una ontología con conocimiento sobre CBR reutilizable para distintos dominios y aplicaciones, que incluye terminología general, representación explícita de tareas CBR y una biblioteca de PSMs reutilizables para resolverlas que:
  - Incluye métodos representativos de los sistemas CBR intensivos en conocimiento.
  - Saca partido del conocimiento terminológico del dominio explícitamente representado y hace inferencias sobre él basándose en los mecanismos de razonamiento de las DLs.
  - Es limitada pero fácilmente extensible ya que permite añadir nuevos métodos sin influir a las aplicaciones previamente diseñadas.
- Definición de una metodología de diseño de aplicaciones CBR intensivas en conocimiento basada en desvincular los casos y el conocimiento del dominio de las tareas y los métodos CBR, su concreción en el sistema COLIBRI y la prueba de su viabilidad con el desarrollo de varias aplicaciones.

Para llevar a cabo este trabajo hemos:

- Identificado terminología estándar sobre CBR, inherente a cualquier sistema CBR pero independiente de cualquier dominio y aplicación, y hemos formalizado dicha terminología en un sistema de lógica descriptiva.
- Identificado las tareas que se resuelven en los sistemas CBR y las hemos representado de forma explícita como parte de la ontología CBR.
- Estudiado trabajos que proponen extender los procesos de razonamiento con casos con razonamientos basados en conocimiento adicional, incluyendo los que han aplicado mecanismos de las DLs para sistemas CBR, y hemos comprobado su adecuación.

- Puesto de manifiesto la utilidad de los mecanismos de inferencia de las DLs para construir sistemas CBR en general y sistemas KI-CBR en particular.
- Aplicado las lecciones aprendidas para definir y formalizar una biblioteca de métodos que resuelven las tareas CBR identificadas, usando terminología estándar sobre CBR que permitan su reutilización en distintos contextos.
- Identificado las condiciones necesarias para que los métodos sean aplicables y las hemos representado en forma de requisitos de aplicabilidad de los distintos métodos.
- Identificado distintas formas de configuración de los métodos que permitan reutilizarlos de una forma flexible y ajustar su comportamiento a las características de la aplicación y dominio concretos.
- Propuesto una arquitectura en varias capas, que disgrega el conocimiento presente en los sistemas CBR, principalmente los casos, el conocimiento terminológico del dominio, las tareas que se resuelven y los métodos que lo hacen.
- Promovido la reutilización de cada uno de los componentes anteriores que pueden provenir de distintas fuentes. Para ello hemos estudiado si el conocimiento presente en las bibliotecas de ontologías del dominio es adecuado para su uso en los procesos involucrados en el CBR.
- Concluido que el conocimiento terminológico definido en las ontologías es adecuado para su uso en CBR si se lleva a cabo un proceso de integración de conocimiento que permita *marcar* los términos del dominio con la terminología CBR de forma que puede ser usada por los métodos de la biblioteca de CBROnto.
- Propuesto un proceso de integración basado en la clasificación de las DLs que aprovecha sus mecanismos de herencia y propagación de restricciones.
- Aplicado las ideas anteriores para implementar COLIBRI, un entorno de desarrollo de aplicaciones KI-CBR, basado en CBROnto, que ocupa el núcleo básico de la arquitectura del sistema. Lo que COLIBRI aporta respecto a otras herramientas es el uso intensivo de conocimiento del dominio y la disposición de una biblioteca de métodos que permite al diseñador de una aplicación estudiar el comportamiento de distintos métodos que sacan partido de este conocimiento a través de los mecanismos de razonamiento de las DLs. Otra ventaja de COLIBRI es el uso de casos complejos y estructurados.
- Propuesto e incluido en COLIBRI una metodología de diseño de aplicaciones KI-CBR que se basa en:
  - Reutilizar conocimiento del dominio de una biblioteca de ontologías.
  - Reutilizar terminología, tareas y métodos CBR.
  - Integrar el conocimiento del dominio con la terminología de CBROnto para que los métodos genéricos puedan hacer uso intensivo de dicho conocimiento.
  - Configurar la resolución de las tareas, seleccionando qué tareas se resolverán y qué métodos se usarán para ello, y configurando los métodos a través de los requisitos que habíamos identificado previamente.
- Definido un proceso de selección de métodos aplicables basado en:

  - La definición de un vocabulario de cualificadores de conocimiento.
  - La representación explícita de las características del contexto de aplicación concreto y de los requisitos de aplicabilidad de los métodos usando este vocabulario.
  - El uso de los mecanismos de razonamiento de las DLs, en concreto la subsunción y el reconocimiento de instancias, para determinar la aplicabilidad de un método en un contexto.
- Probado la viabilidad de nuestra aproximación con el desarrollo de varias aplicaciones ejemplo.

Asimismo nuestro trabajo ha puesto de manifiesto algunas limitaciones a tener en cuenta que se resumen en el siguiente apartado.

3. Limitaciones

La necesidad de disponer de un modelo explícito de conocimiento del dominio es un problema inherente a los KBSs en general. Los beneficios obtenidos con el uso del conocimiento y las capacidades de razonamiento adquiridas en una aproximación justifican el esfuerzo de adquisición de conocimiento asociado con este tipo de sistemas. En nuestra propuesta las ontologías juegan un papel importante en el proceso de adquisición de conocimiento y aunque su uso supone muchas ventajas también hemos encontrado algunos inconvenientes:

- La adecuación de nuestra propuesta depende directamente de la existencia de una biblioteca de ontologías y de la existencia de una o varias ontologías sobre el dominio de aplicación. La no existencia requiere la formalización manual de una base de conocimiento sobre el dominio.
- Suponiendo que encontremos ontologías adecuadas para el dominio de aplicación, en dominios complejos o con distintas facetas puede ser necesario integrar distintas fuentes de conocimiento. Esto supone solventar problemas como la definición de términos con distinto nivel de abstracción.
- Además, la compartición de conocimiento plantea problemas a nivel de implementación debido a la diversidad de formalismos de representación de conocimiento. El uso de herramientas (como OntologyServer) y los estándares emergentes como DAML+OIL pueden ayudar a automatizar el proceso de traducción. En concreto, en el estado actual la implementación de CBROnto y COLIBRI es dependiente de LOOM –aunque la conceptualización es independiente y se podría reimplementar en otro sistema de DL– y requiere la disponibilidad de una biblioteca de ontologías formalizadas en dicho lenguaje o el uso de traductores adecuados.

Respecto a la adquisición de casos, en una aplicación diseñada utilizando COLIBRI los casos deben definirse en base a la terminología disponible sobre el dominio y a la terminología de CBROnto. Existiría un problema claro de integración de conocimiento si quisiéramos incorporar casos existentes en algún formato electrónico.

La incorporación a COLIBRI de traductores que acepten algunos formatos estándar (como XML) podría aliviar este problema. Sin embargo existe un problema adicional de integración terminológica que se debe resolver de forma no automática, ya que los métodos de CBROnto están definidos en base a cierta terminología estándar CBR y no pueden razonar con casos que no utilicen esta terminología, directamente o a través de la clasificación de

términos. Es decir para los casos ocurrirá lo mismo que con el conocimiento general sobre el dominio: para que los métodos puedan sacar partido de este conocimiento (general o casos) debe estar integrado con la terminología que "esperan" los métodos.

Respecto al proceso de diseño de aplicaciones CBR hemos observado, a través de los ejemplos, que la configuración de los métodos requiere en ocasiones que el diseñador conozca de forma bastante detallada la estructura del conocimiento del dominio. Aunque esto puede requerir un esfuerzo importante para el diseñador, el uso de interfaces gráficas de visualización y de ayuda para la configuración de los métodos aliviará el problema. Además la posibilidad de prototipar y probar inmediatamente los resultados obtenidos facilita el proceso. Intentaremos resolver, al menos en lo posible, estas limitaciones con las líneas de trabajo futuro que se describen en el siguiente apartado.

4. Trabajo Futuro

Finalizamos esta memoria con la presentación de la líneas de continuación del trabajo realizado hasta el momento.

El primer aspecto en el que incidiremos es en el desarrollo de una interfaz mejor, más sencilla y amigable que ayude al diseñador en la tarea de desarrollo. En la versión actual COLIBRI no dispone de una interfaz gráfica aunque un prototipo de la misma está siendo implementado en Java con mecanismos de intercambio de información en CORBA.

Para solventar la dependencia actual de un formalismo de representación concreto como LOOM, nos planteamos el uso de estándares de representación de conocimiento e intercambio de información como XML o DAML+OIL.

El conocimiento que hemos incluido en CBROnto es limitado, y aunque la estructura básica actual sirve de esqueleto integrador, se puede incluir nuevo conocimiento, principalmente en forma de instancias. Durante el diseño de una aplicación el nuevo conocimiento representa el conocimiento específico de la aplicación que incluirá el diseñador de la aplicación. En este apartado no nos referimos a este tipo de conocimiento sino a la posibilidad de extender la ontología con conocimiento reutilizable adicional. Por ejemplo, una de las facetas en la que sería deseable la inclusión de nuevo conocimiento es en forma de nuevas tareas y métodos.

Por tanto, además de las líneas de trabajo anteriores, recogemos en una línea de trabajo común las posibles extensiones a CBROnto, entre las que nos plantearemos las siguientes:

- Incluir métodos de aprendizaje más sofisticados que saquen partido de las interacciones del usuario. Relacionado con este punto nos plantearemos incluir métodos de olvido de los casos que no resulten útiles. Este estudio de la utilidad de los casos permitirá que el sistema decida si un nuevo caso debe o no ser aprendido.
- Añadir métodos que razonen con otros tipos de conocimiento además del terminológico. En particular, nos hemos planteado la representación y uso de reglas complejas, aunque habría que estudiar la adecuación de las DLs como formalismo de representación de las mismas. En la versión actual, los mecanismos de complección de instancias permiten trabajar con reglas simples de pertenencia a conceptos, como hemos hecho para representar las reglas de dependencia extraídas del AFC.
- Incluir métodos genéricos que puedan ser *especializados* en distintas aplicaciones. En la versión actual los métodos se pueden configurar pero no se permiten variaciones

adicionales a no ser que se modifique el código Lisp correspondiente a la especificación operacional de los métodos. Como trabajo futuro nos planteamos incluir nuevos elementos en el lenguaje de descripción de métodos que faciliten la definición de clases de métodos con distintas instanciaciones, así como proporcionar un entorno de desarrollo con mecanismos de trazas y que facilite la extensión o modificación de los métodos incorporados. En esta línea se pueden permitir algunas adaptaciones simples de un método genérico a las circunstancias específicas de nuestra aplicación, por ejemplo, variaciones automáticas en un método dependiendo de la profundidad de la jerarquía o del número de casos, información que podemos obtener de forma directa por clasificación o reconocimiento de instancias.

- La necesidad de la inclusión de nuevas tareas encaja con los recientes trabajos en la comunidad de CBR, que han ampliado el ciclo CBR clásico para que incluya nuevas tareas. En concreto resultan muy relevantes las aportaciones relativas a métodos de mantenimiento de bases de casos [Smyth99] [Leake *et al.* 01]. En estos trabajos se reconoce la existencia de más de cuatro tareas básicas en el ciclo CBR, y proponen la inclusión de tareas adicionales relacionadas con el mantenimiento. En la versión actual de CBROnto no se incluye la representación explícita de métodos de mantenimiento aunque prevemos hacerlo en un futuro inmediato. La extensión del conocimiento de nuevas tareas al esqueleto general resulta muy sencilla. Por ejemplo, incluir la tarea de mantenimiento supone únicamente añadir un nuevo concepto y un individuo canónico para representar la tarea. La tarea de mantenimiento podría entenderse como una tarea al mismo nivel que la tarea principal CBR, o como una subtarea de la anterior. La resolución de la tarea de mantenimiento se puede realizar periódicamente o bajo demanda del usuario y la ventaja de nuestro proceso de resolución genérico es que accederá al conocimiento explícitamente representado para obtener los métodos que resuelven la tarea y aplicará uno de los métodos de mantenimiento de forma análoga al proceso de resolución descrito para el resto de las tareas de CBROnto.
- La consideración de casos a distinto nivel de abstracción para incluir, por ejemplo, la distinción entre casos abstractos, casos concretos y casos jerárquicos [Branting&Aha95][Bergmann&Wilke96]. Esta aproximación encaja con nuestro esquema ya que la estructura de casos resultante se basa en jerarquías de generalización/especialización. Los procesos de búsqueda asociados con este tipo de jerarquías comienzan en el punto más general de la jerarquía y progresan hacia nodos más específicos sólo si es posible el encaje (*matching*) al nivel general. La búsqueda se poda en cualquier rama de la jerarquía en la que el encaje no sea posible. De esta forma los casos específicos de la memoria sólo pueden ser recuperados cuando sus abstracciones en la jerarquía encajen con la consulta. Estas ideas también las comparte el sistema PARIS [Bergmann&Wilke96] en el que se define un método de adaptación basado en la abstracción y refinamiento de casos a distintos niveles de abstracción.
- Por último, añadir métodos de reutilización de múltiples casos. La recuperación no tiene por qué recuperar un único caso y la adaptación debería ser capaz de aprovecharlos lo que se puede afrontar de varias formas (varios métodos):
  - Usando CBR Jerárquico que permite la reutilización explícita de descomposiciones de problemas, para recuperar y adaptar los subproblemas.

– Combinar los *n* casos recuperados si se incluye la representación explícita de dependencias y restricciones entre distintas partes de las soluciones y entre descripciones y soluciones.

Esperamos que los aspectos anteriores ayuden en la consecución del siguiente objetivo que nos planteamos y que consiste en probar, verificar y refinar las definiciones de CBROnto haciendo uso de las aportaciones que obtengamos de su uso en sistemas reales. En la situación actual CBROnto se encuentra en una fase de desarrollo más "teórica" en la que se ha identificado una terminología descriptiva de distintos aspectos, se ha utilizado en los métodos de la biblioteca y ha sido el núcleo de COLIBRI en todas las aplicaciones diseñadas. Sin embargo nos gustaría recibir aportaciones a partir del uso de CBROnto (en el contexto de COLIBRI o en otro) en sistemas reales externos a nuestro grupo de investigación que realimente las definiciones tanto para aceptarlas o mejorarlas, como para rechazarlas.

Nuestro trabajo se beneficiaría en gran medida del hecho de disponer de una batería de sistemas CBR diseñados con CBROnto, que sirva como punto de partida para definir una base de casos de diseño de sistemas CBR que ayude en el diseño de nuevos sistemas. Es decir, usar CBR en sí mismo para resolver la tarea de diseñar un sistema CBR con CBROnto, de forma que se aprovechen configuraciones previas de sus tareas y métodos, o esquemas de representación o indexación de casos, que hayan resultado exitosas en el diseño de sistemas CBR previos usando CBROnto. El marco de CBROnto permite fácilmente la gestión de estos casos, siendo el principal escollo el poder disponer de un número suficientemente significativo de sistemas diseñados.

Bibliografía

[Aamodt91] Aamodt, A., 1991. *knowledge-intensive, integrated approach to problem solving and sustained learning*, Ph.D (Dr.Ing.) dissertation, University of Trondheim, Norwegian Institute of Technology, Department of Computer Science.

[Aamodt93] Aamodt, A., 1993. "Explanation-driven case-based reasoning". *Topics in Case-based reasoning*. Wess, S., Althoff, K.-D., & Richter, M., (Eds.). Springer Verlag, pp. 274-288.

[Aamodt&Plaza94] Aamodt A., Plaza E., 1994. "Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches". *Artificial Intelligence Communications*, 7 (1), pp. 39-59.

[Aamodt00] Aamod, A., 2000. "Modeling the knowledge contents of CBR systems". Workshop on CBR Authoring Support Tools. Naval Research Lab Technical Note AIC-01-003, pp. 32-37.

[Aha91] Aha, D., 1991. "Case based learning algorithms". Procs. of the CBR Workshop. Bareiss R. (Ed.). Morgan Kaufmann, pp. 147-158.

[Aha97] Aha, D. (Ed.), 1997. Lazy Learning. (Reprinted from: Artificial *Intelligence Review*,11,1-5) Kluwer Academic Publishers.

[Aha00] Aha, D., 2000. "Interactive Case-Based Reasoning: Influences, Utility, and Outlook in an Applied World", Procs of the International Conference on Industrial and Engineering Applications of Artificial Intelligence & Expert Systems. New Orleans.

[Alberts93] Alberts, D., 1993. *YMIR: an ontology for engineering design*. University of Twente, PhD thesis.

[Althoff *et al.* 95] Althoff K.-D., Auriol E., Barletta R., Manago M., 1995. "A Review of Industrial Case-Based Reasoning Tools" *AI Intelligence*, Oxford UK.

[Arcos97] Arcos, J.Ll., 1997. *The Noos Representation Language*, Tesis Doctoral. Universidad Politécnica de Cataluña, Dep. de lenguajes y sistemas informáticos.

[Arcos&Plaza93] Arcos, J.Ll., & Plaza, E.,1994. "A reflective Architecture for Integrated Memory-based Learning and Reasoning", *Topics in Case-based reasoning* (EWCBR'93). Wess, S., Althoff, K.-D., & Richter, M., (Eds.). Springer Verlag, LNAI 837, pp.289-300.

[Armengol&Plaza93] Armengol, E., & Plaza, E., 1993. "A Knowledge Level Model of Knowledge-Based Reasoning". *Topics in Case-Based Reasoning*, Wess, S, Althoff, K., & Richter, M. (Eds.). Springer-Verlag LNAI 837, pp. 53-64.

[Armengol&Plaza01] Armengol, E., & Plaza, E., 2001. "Similarity Assessment for Relational CBR". *CBR Research and Development* (ICCBR'01). Aha, D. & Watson, I. (Eds.). Springer-Verlag LNAI 2080, pp. 44-58.

[Arpirez *et al.* 98] Arpirez, J., Gómez-Pérez, A., Lozano, A., & Pinto S., 1998. "(onto)2agent: an ontology based www broker to select ontologies". Procs. of the ECAI'98 Workshop of applications of ontologies and problem-solving methods, pp. 16-24.

[Arpirez *et al.* 01] Arpirez, J., Corcho, O., Fernández-López, M, & Gómez-Pérez, A., 2001. "WebODE: a scalable workbench for ontological engineering" Procs. of the 1st Int. Conference on Knowledge Capture (K-CAP'01), Canada. ACM Press New York, pp. 6-13.

[Ashley91] Ashley, K., 1991. Modeling legal arguments: Reasoning with cases *and hypotheticals*. MIT Press, Bradford Books, Cambridge.

[Ashley&Aleven93] Ashley, K.D., & Aleven, V., 1993. "A logical representation for relevance criteria". *Topics in Case-based reasoning* (EWCBR'93). Wess, S., Althoff, K., & Richter, M., (Eds.). Springer Verlag, LNAI 837, pp. 338-352.

[Baader&Hollunder91] Baader, F., & Hollunder, B., 1991. "KRIS: Knowledge representation and inference system". *SIGART Bulletin* 2 (3), pp. 8-14.

[Baader&Sattler98] Baader, F., & Sattler, U., 1998. "Description Logics with Concrete Domains and Aggregation", *Procs. of the 13th European Conference on Artificial Intelligence (*ECAI'98*)*, Prade, H., (Ed.), John Wiley & Sons, ltd., 1998. Brighton, UK.

[Baader *et al.* 99] Baader, F., Molitor, R., & Tobies, S., 1999. "Tractable and Decidable Fragments of Conceptual Graphs". Procs. of the ICCS'99.7

[Baader *et al.* 02] Baader, F., Calvanese, D., Mc Guinness, D., Nardi, D., Patel-Schneider, P., (Eds.), 2002. *The Description Logics Handbook: Theory, Implementations and Applications*, Cambridge University Press.

[Bareiss88] Bareiss, E. R.,1988. *PROTOS: A Unified Approach to Concept Representation, Classification, and learning.* Ph.D. thesis, Department. of Computer Science, University of Texas.

[Bareiss *et al.* 90] Bareiss, R., Porter . B., & Holte, R., 1990. "Concept Learning and Heuristic Classification in Weak-Theory Domains", *Artificial Intelligence Journal*, v45 (nos. 1-2), pp. 229-264.

[Barletta&Hennessy89] Barletta R., & Hennessy D., 1989. "Case Adaptation in autoclave layout design". Procs of the Workshop on case-based reasoning *(DARPA)*, Morgan Kaufmann, Florida, pp. 203-207.

[Bateman90] Bateman, J.A., 1990. "Upper modeling: organizing knowledge for natural language processing". *5th. International Workshop on Natural Language Generation*, Pittsburgh, PA.

[Beghardt *et al.* 97] F. Gebhardt, F., Voß, A., Gräther, W., & Schmidt-Belz, B. (Eds.), 1997. *Reasoning with Complex Cases* Kluwer Academic Publishers, Boston. International Series in Engineering and Computer Science Vol. 393.

[Benjamins95] Benjamins, R., 1995. "Problem Solving Methods for diagnosis and their role in knowledge acquisition", *International Journal of Expert Systems: Research&Applications*, 8 (2), pp. 93-120.

[Benjamins *et al.* 96] Benjamins, R., Fensel, D., & Straatman, R., 1996. "Assumptions of problem solving methods and their role in knowledge engineering". Procs. of the European Conference on Artificial Intelligence (ECAI'96), Wahlster, W. (Ed.), Wiley &Sons, Ltd., pp. 408-412.

[Benjamins *et al.* 99] Benjamins, R., Wielinga, B., Wielemaker, J., & Fensel, D., 1999. "Brokering Problem-Solving Knowledge at the Internet". *Knowledge Acquisition, Modeling and Management* (EKAW'99), Fensel *et al.* (Eds.), LNAI 1621, Springer-Verlag.

[Benjamins&Fensel98] Benjamins, V. R, & Fensel, D., 1998. "Special issue on problem-solving methods". *International Journal of Human-Computer Studies*, 49(4), Guest Editorial, pp. 305-313.

[Bergmann&Wilke96] Bergmann, R. & Wilke, W., 1996. "PARIS: Flexible plan adaptation by abstraction and refinement." Procs. of the *ECAI'96 Workshop on Adaptation in Case-Based Reasoning.*

[Bergmann *et al.* 96] Bergmann, R., Wilke, W., Vollrath, I., Wess, S., 1996. "Integrating General Knowledge with Object-Oriented Case Representation and Reasoning". *4th German Workshop: Case-Based Reasoning - System Development and Evaluation*, H.-D. Burkhard & M. Lenz (Hrsg.), Informatik-Berichte Nr. 55, Universität Berlin, pp. 120-127.

[Bergmann *et al.* 97] Bergmann R., Wilke W., Althoff K.-D., Breen S. & Johnston R., 1997. "Ingredients for Developing a Case-Based Reasoning Methodology". 5th German Workshop on CBR (GWCBR'97).

[Bergmann&Stahl98] Bergmann, R., Stahl, S., 1998. "Similarity Measures for Object-Oriented Case Representations". *Advances in Case-Based Reasoning* (EWCBR'98), Smyth, B., & Cunningham, P., (Eds.) Springer, LNAI 1488, pp. 25-36.

[Bergmann&Wilke98] Bergmann, R., & Wilke, W., 1998. "Towards a New Formal Model of Transformational Adaptation in Case-Based Reasoning". *Procs. of the 13th European Conference on Artificial Intelligence* (ECAI'98), Prade, H., (Ed.), John Wiley & Sons, ltd., 1998. Brighton, UK.

[Bergmann *et al.* 99] Bergmann, R., Breen, S., Goker, M., Manago, M. & Wess, S., (Eds.), 1999. *DevelopingIndustrial Case-Based Reasoning Applications : The INRECA Methodology*.Springer, LNAI 1612.

[Beys *et al.* 96] Beys, P., Benjamins, R., Van Heijst, G., 1996. "Remedying the Reusability—Usability Tradeoff for Problem Solving Methods", Procs of Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop, SRDG Publications, university of Calgary. http://ksi.cpsc.ucalgary.ca:80/KAW/KAW96/KAW96Proc.html

[Borgida92] Borgida, A., 1992. "Description Logics are not just for the Flightless-Birds: A new Look at the Utility and Foundations of Description Logics". Tech. Rep., Dept. of Computer Science, Rutgers University, New Brunswick, NJ.

[Borgida95] Borgida, A., 1995. "Descriptions Logics in Data Management". *Knowledge and Data Engineering*, 7 (5), pp. 671-682.

[Borgida96] Borgida, A., 1996. "On the Relative Expresiveness of Description Logics and Predicate Logics". *Artificial Intelligence* 82, pp. 353-367.

[Borst97] Borst, W.N., 1997. *Construction of Engineering Ontologies*. PhD Thesis, University of Twente.

[Brachman&Levesque84] Brachman, R., Levesque, H.J., 1984. "The tractability of subsumption in frame-based description languages". Procs. of AAAI'84, pp. 34-37.

[Brachman&Schmolze85] Brachman, R., & Schmolze, J. G., 1985. "An overview of the KL-ONE knowledge representation system". *Cognitive Science*, 9 (2), pp. 171-216

[Brachman *et al.* 85] Brachman, R., Gilbert, V., & Levesque, H., 1985. "An essential hybrid reasoning system: knowledge and symbol level accounts in KRYPTON. Procs of the International Joint Conference on AI (IJCAI'95), California, pp. 532-539.

[Brachman&Levesque87] Brachman, R. & Levesque, H., 1987. "Expressiveness and Tractability in Knowledge Representation and Reasoning". *Computer Intelligence*, 3, pp. 78-93.

[Brachman *et al.* 91] Brachman R. J., McGuinness D. L., Patel-Schneider, P. F., Resnick, L. A., & Borgida, A., 1991. "Living with CLASSIC: When and How to Use a KL-ONE-Like Language". *Principles of Semantic Networks*. Sowa, J. F.(Ed.), Morgan Kaufmann., pp. 401-456.

[Branting&Aha95] Branting, L. K. & Aha, D., 1995. "Stratified CBR: Reusing Hierarchical Problem Solving Episodes". Procs of the *14th* International Joint Conference on Artificial Inteligence (IJCAI'95), Montreal, Canada, Morgan Kaufmann, pp. 384-390.

[Breuker&vandeVelde94] Breuker, J., & Van de Velde, W., 1994 *CommonKADS Library for Expertise Modelling*, IOS PRess, Amsterdam.

[Bridge98] Bridge, D.G., "Defining and Combining Symmetric and Asymmetric Similarity Measures". *Advances in Case-Based Reasoning* (EWCBR'98), Smyth, B., & Cunningham, P., (Eds.) Springer, LNAI 1488, pp. 52-63.

[Brill93] Brill, D., 1993. Loom Reference Manual for Loom version 2.0.

[Brown93] Brown, M., 1993. *A Memory Model for Case Retrieval by Activation Passing*. Ph.D. thesis, Department of Computer Science, University of Manchester, Technical Report UMCS-94-2-1.

[Brüninghaus&Ashley01] S. Brüninghaus & K. D. Ashley , 2001. "The Role of Information Extraction for Textual CBR". *Case-Based Reasoning Research and Development* (ICCBR'01). Aha, D., & Watson, I. (Eds.). Springer, LNAI 2080, pp. 74-89.

[Bunke&Messmer93] Bunke, H. & Messmer, B.T., 1993. "Similarity Measures for Structured Representations". *Topics in Case-based reasoning* (EWCBR'93). Wess, S., Althoff, K.-D., & Richter, M., (Eds.). Springer Verlag, LNAI 837, pp. 106-118.

[Bylander&Chandrasekaran88] Bylander, T., & Chandrasekaran, B., 1988. "Generic tasks in knowledge-based reasoning: The right level of abstraction for knowledge acquisition". *Knowledge Acquisition for Knowledge Based*

Systems, Gaines, B., & Boose, J., (Eds.) Vol 1. Academic Press, London,. pp. 65-77.

[Calvanese *et al.* 98a] Calvanese, D., Lenzerini, M., & Nardi, D., 1998. `Description Logics for Conceptual Data Modeling', *Logics for Databases and Information Systems*, Chomicki, J. & Saake, G. (Eds.), Kluwer.

[Calvanese *et al.* 98b] Calvanese, D., De Giacomo, G., M. Lenzerini, D. Nardi, & R. Rosati, 1998. "Description Logic framework for information integration". Proc. of KR'98, pp. 2-13.

[Calvanese *et al.* 99] Calvanese, D., De Giacomo, G., & M. Lenzerini, 1999. "Reasoning in expressive Description Logics with fixpoints based on automata on infinite trees". Procs. of the 16th International Joint Conference on AI (IJCAI'99), Stockholm, Sweden, pp. 84-89.

[Calvanese *et al.* 01] Calvanese, D., De Giacomo, G., & Lenzerini, M., 2001. "Description Logics: Foundations for Class-based Knowledge Representation". *Proc. of the 17th IEEE Sym. on Logic in Computer Science (LICS).*

[Carbonell83] Carbonell, J. G. 1983, "Learning by Analogy: Formulating and Generalising Plans from Past Experience". *Machine Learning*, Vol. 1, Michalski, R., & Carbonell, J., Mitchell, T., (Eds.) Morgan Kaufmann, pp. 137-161.

[Chandrasekaran86] Chandrasekaran B., 1986. “Generic Tasks for knowledge-based reasoning: High-level building blocks for expert system design”, *IEEE Expert*, 1(3), pp. 23-30.

[Chandrasekaran87] Chandrasekaran B., “Towards a functional architecture for intelligence based on generic information processing tasks”. Procs of the International Joint Conference on Artificial Inteligence (IJCAI'87), Milan, Italy, pp. 1183-1192.

[Chandrasekaran90] Chandrasekaran B., 1990. “Design problem solving: A task analysis”. AI Magazine, (11), pp. 59-71.

[Chandrasekaran *et al.* 92] Chandrasakaran, B., Johnson, J.R., & Smith, J.W., 1992. “Task-structure analysis for knowledge modelling”, *Communication of the ACM*, 35(9), pp. 124-137.

[Chandrasekaran *et al.* 98] Chandrasekaran, B., Josephson, J.R, & Benjamins, R., 1998. “Ontology of tasks and Methods”, *AAAI Spring Symposim*. Also in Procs of the 11th Workshop on Knowledge Acquistion, Modeling, and Management, (KAW'98).

[Chandrasekaran *et al.* 99] Chandrasekaran, B., Josephson, J.R, & Benjamins, R., 1999. “Ontologies: What are they? Why do we need them?”, IEEE Intelligent Systems and Their Applications, 14(1), pp. 20-26. Special Issue on Ontologies.

[Cohen *et al.* 92] Cohen, W.W., Borgida, A., & Hirsh, H., 1992. “Computing least common subsummers in description logics”, in 10th National Conference of the American Association for Artificial Intelligence, pp. 754-760.

[Cohen *et al.* 99] Cohen, P., Chaudhri, V., Pease, A., & Schrag, R., 1999. “Does Prior Knowledge Facilitate the Development of Knowledge Based Systems?”. Procs of the AAAI'99.

[Corcho&Gómez-Perez00a] Corcho, O., & Gómez-Pérez, A., 2000. "Evaluating Knowledge Representation and Reasoning Capabilities of Ontology Specification

Languages" Procs of the ECAI'00 Workshop on Applications of Ontologies and Problem Solving Methods. Berlin. Germany.

[Corcho&Gómez-Perez00b] Corcho, O., & Gómez-Pérez, A., 2000. "A Roadmap to Ontology Specification Languages". Procs of the 12th International Conference on Knowledge Engineering and Knowledge Management (EKAW'00). Juan-les-Pins. France.

[Corcho&Gómez-Perez00c] Corcho, O., Gomez-Peréz, A., 2000. "Ontology Specification Languages for the Semantic Web". *IEEE Intelligent Systems*.

[Coupey *et al.* 98] Coupey, P., Fouqueré, C., & Salotti, S., 1998. "Formalizing Partial Matching and Similarity in CBR with a Description Logic". *Applied Artificial Intelligence*, 12 (1), pp. 67-108.

[Davey&Priestley90] Davey, B.A., & Priestley, H.A., 1990. "Introduction to lattices and order". *Cambridge Mathematical Textbooks*.

[Davies&Rusell87] Davies, T.R., & Russell, S.J., 1987. "A logical approach to Reasoning by analogy" Procs of the International Joint Conference on Artificial Inteligence (IJCAI'87), Milan, Italy.

[Devanbu *et al.* 91] Devanbu, P., Brachman, R., Selfridge, P., & Ballard, B., 1991. "LaSSIE - A Classification-Based Software Information System". *Communications of the ACM*, Mayo.

[Díaz&González00a] Díaz-Agudo, B., & González-Calero, P.A., 2000. "An Architecture for Knowledge Intensive CBR Systems". *Advances in Case-Based Reasoning* (EWCBR'00), Blanzieri, E., Portinale, L., (Eds.), Springer, LNAI 1898, pp. 37-48.

[Díaz&González00b] Díaz-Agudo, B., & González-Calero, P.A., 2000. "Formal Concept Analysis as a Support Technique for CBR". *Procs. of the Twentieth SGES International Conference on Knowledge Based Systems and Applied Artificial Intelligence* (ES'00) Cambridge, UK. Bramer, M., Preece, A., & Coenen, F. (Eds.). Springer Verlag.BCS conference series, pp. 117-128.

[Díaz&González01a] Díaz-Agudo, B., & González-Calero, P.A., 2001. "Formal Concept Analysis as a Support Technique for CBR". *Knowledge-Based Systems*, 14 (3-4), June, Elsevier. (ISSN:0950-7051), pp. 163-172.

[Díaz&González01b] Díaz-Agudo, B., & González-Calero, P. A., 2001. "A Declarative Similarity Framework for Knowledge Intensive CBR". *Case-Based Reasoning Research and Development* (ICCBR'01), Aha, D., Watson, I., (Eds.), Springer LNAI 2080, pp. 158-172.

[Díaz&González01c] Díaz-Agudo, B., & González-Calero, P. A., 2001. "Classification Based Retrieval Using Formal Concept Analysis". *Case-Based Reasoning Research and Development* (ICCBR 01), Aha, D., Watson, I., (Eds.) Springer LNAI 2080, pp. 172-187.

[Díaz&González01d] Díaz-Agudo, B., & González-Calero, P. A., 2001. "A Similarity Representation Framework for Knowledge Intensive CBR". In Schnurr, H., Staab, S., Studer, R., Stumme, G., Sure, Y., (Eds.): Professionelles WissensManagement (Proc. of the German Workshop on Case-Based Reasoning, GWCBR'01). Springer-Verlag.

[Díaz&González01e] Díaz-Agudo, B., & González-Calero, P. A., 2001. "Knowledge Intensive CBR Made Affordable". In Weber, R., Gresse von Wangenheim, C., (Eds.): Procs. of the Workshop Program at

ICCBR'01, *Technical Note AIC-01-003, Navy Center for Applied Research in Artificial Intelligence*, Washington, DC, USA.

[Díaz&González01f] Díaz-Agudo, B., & González-Calero, P. A., 2001. "Integración de ontologías en los sistemas de razonamiento basado en casos". Actas de la IX Conferencia de la Asociación Española para la Inteligencia Artificial (CAEPIA'01), pp. 501-510

[Díaz&González01g] Díaz-Agudo, B., & González-Calero, P. A., 2001. "Knowledge Intensive CBR through Ontologies". Proc. of the 6th UK Workshop on Case-Based Reasoning (UKCBR'01) , Lees, B., (Ed.)., University of Paisley, pp. 43-52.

[Díaz&González02] Díaz Agudo, B., & González Calero, P.A., 2002. "CBROnto: a task/method ontology for CBR". *Procs. of the 15th International FLAIRS'02 Conference* (Special Track on Case-Based Reasoning). Haller, S., &, Simmons, G., (Eds.). AAAI Press, pp. 101-106.

[Díaz *et al.* 02] Díaz-Agudo, B., Gervás, P. & González-Calero, P., 2002. "Poetry Generation in COLIBRI". Procs 6th European Conference on Case Based Reasoning, Aberdeen, Scotland, 4-7 September 2002 (ECCBR'02). Springer. To appear.

[Donini *et al.* 95] Donini, F., Lenzerini, M., Nardi, D., & Nutt. W., 1995. "The complexity of concept languages". Research Report RR-95-07. DFKI (Deutsches Forschungszentrum für Künstiliche Intelligenz).

[Donini *et al.* 96] Donini, F., Lenzerini, M., Nardi, D., & Schaerf, A., 1996. "Reasoning in Description Logics". *Principles of Knowledge Representation and Reasoning*, Brewka; G. (Ed.), CLSI Publications, pp. 193-238.

[Donini *et al.* 99] Donini, F., Lenzerini, M., Nardi, D., & Nutt, W., 1999. "Tractability and Intractability in DLs". Draft que extiende y corrige el artículo de los mismos autores y título en Procs. of IJCAI'91. (ftp://ftp.dis.uniroma1.it/pub/ai/papers/dlnn99d.ps.gz)

[Farquhar *et al.* 95] Farquhar, A., Fikes, R., Pratt, W., & Rice, J., 1995. Collaborative ontology construction for information integration. *Technical Report KSL-95-63, Stanford University Knowledge Systems Laboratory*.

[Farquhar *et al.* 97] Farquhar, A., Fikes, R., & Rice, J., 1997. The ontolingua server: a tool for collaborative ontology construction. *International Journal of Human-Computer Studies*, 46(6), pp. 707-728.

[Fensel *et al.* 97] Fense,l D., Motta, E., Decker, S., & Zdrahal, Z., 1997. "Using Ontologies for Defining Tasks, Problem-Solving Methods, and Their Mappings", Knowledge Acquisition, Modeling, and Management, E. Plaza, & V.R. Benjamins (Eds.), Springer-Verlag, Berlin, pp. 113-128.

[Fensel *et al.* 98a] Fensel, D., Benjamins, R., Gaspari, S., Decker, R., Groenboom, W., Grosso, M, Musen, M., Motta, E., Plaza, E., Schriber, G., Studer, R. & Wielinga, B., 1998. "The component model of UPML in a nutshell". Procs. of the International Workshop on Knowledge Acquisition KAW'98, Canadá.

[Fensel *et al.* 98b] Fensel, D., Decker, S., Erdmann, M. & Studer, R., 1998. "Ontobroker: the very high idea". *Procs. of the 11th International FLAIRS Conference (FLAIRS'98)*. AAAI Press.

[Fensel&Motta98] Fensel, D., & Motta, E., 1998. "Structured Development of Problem Solving Methods". Procs of the 11th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW98), Banff, Canada.

[Fensel *et al.* 99] Fensel, D., Benjamins, R., Motta, E., & Wielinga, B., 1999. "UPML: A Framework for knowledge system reuse". *Procs of the International Joint Conference on Artificial Intelligence (IJCAI-99)*, Sweden.

[Fensel01] Fensel, D. (Ed.), 2001. *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer Verlag.

[Fernández *et al.* 99] Fernández, M., Gómez-Pérez, A., Pazos, J., & Pazos, A., 1999. "Building a Chemical Ontology using methontology and the ontology desing environment". *IEEE Intelligent Systems and their applications*, 14 (1), pp. 37-45.

[Fox&Leake95] Fox, S., & Leake, D., 1995. "Combining Case-Based Planning and Introspective Reasoning". Procs of the Sixth Midwest Artificial Intelligence and Cognitive Science Conference.

[Fuchs99] Fuchs, B., 1999. "A knowledge-level analysis of adaptation in CBR". *Challenges for Case-Based Reasoning* - Proc. of the ICCBR'99 Workshops, Centre for Learning Systems and Applications - Dept. of Computer Science - University of Kaiserslautern (Informe Técnico LSA-99-03E), pp. III-8-III-15.

[Gangemi *et al.* 96] Gangemi, A., Steve, G., & Giacomelli, F., 1996. "ONIONS: An Ontological Methodology for Taxonomic Knowledge Integration", 1996. P. Van der Ver (Ed.). Procs. of the ECAI'96 Workshop on Ontological Engineering.

[Ganter&Wille97] Ganter, B., & Wille, R., 1997. *Formal Concept Analysis. Mathematical Foundations*. Springer Verlag.

[Gennari *et al.* 94] Gennari, J.H., Tu, S.W., Rothenfluh, T.E., & Musen, M.A., 1994. "*Mapping* Domains to Methods in Support of Reuse". *International Journal of Human-Computer Studies*, 41, pp. 399-424.

[Gennari *et al.* 98] Gennari, J. H., Grosso, W. & Musen, M., 1998. "A Method-Description Language: An Initial Ontology with Examples". Procs of the 11th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW98), Banff, Canada.

[Genesereth&Fikes92] Genesereth, M.R., & Fikes, R.E., 1992. "Knowledge Interchange Formal. Version 3.0. Reference Manual". Technical Report, Logic-92-1, Computer Science Dep., Stanford University.

[Gervás00] Gervás, P., 2000. "An Expert System for the Composition of Formal Spanish Poetry". *Application and Innovations in Intelligent Systems* VIII, Springer Verlag.

[Gervás01a] Gervas, P., 2001. "Automatic Generation of Poetry Using a CBR Approach". Actas de la IX Conferencia de la Asociación Española para la Inteligencia Artificial (CAEPIA 2001).

[Gervás01b] Gervas, P., 2001. "Modeling Literary Style for the Semi-automatic Generation of Poetry". Proceedings of the 8th International Conference on User Modeling, (UM 2001), Sonthofen, Germany, July 2001, Springer Verlag.

[Gil&Melz96] Gil, Y., & Melz, E. 1996. *Explicit representations of problem-solving strategies to support knowledge acquisition*. Procs of the Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop, SRDG Publications, university of Calgary.

[Gil&Ratnakar02] Gil, Y., & Ratnakar, Y., 2002. "A Comparison of (Semantic) Markup Languages". *Procs. of the 15th International FLAIRS'02 Conference*

(Special Track on Semantic Web). Haller, S., &, Simmons, G., (Eds.). AAAI Press, pp. 413-419.

[Goel96] Goel, A., 1996. "Meta-cases: explaining Case-Based Reasoning *Advances in Case-Based Reasoning* (EWCBR'96), Smith I. & Faltings B. (Eds.). Springer-Verlag LNAI 1168, pp. 150-163.

[Gómez *et al.* 98] Gómez-Albarrán, M., González-Calero, P.A., Díaz-Agudo, B.: "Software Design as Framework Reuse: a Knowledge-Based Approach". Proc. 13th European Conference on Artificial Intelligence, John Wiley & Sons, Reino Unido, 1998.

[Gómez *et al.* 99] Gómez-Albarrán, M., González-Calero, P.A., Díaz-Agudo, B. Fernández-Conde, C., 1999. "Modelling the CBR Life Cycle Using Description Logics". *Case-Based Reasoning Research and Development* (ICCBR'99), Althoff, K.D., Bergmann, R., Branting, L.K., (Eds.), Springer, LNAI 1650.

[Gómez-Albarrán00] Gómez-Albarrán, M., 2000. *Aplicación de técnicas de representación de conocimiento a la reutilización de diseño en programación orientada a objetos*. Tesis Doctoral, Dep. de Sistemas Informáticos y Programación, Universidad Complutense de Madrid.

[Gómez-Pérez *et al.* 97] Gómez-Pérez, A., Juristo, N., Montes, C. & Pazos, J., 1997. *Ingeniería del Conocimiento*. Editorial Centro de Estudios Ramón Areces, S.A.

[Gómez-Pérez98] Gómez-Pérez, A.,1998. "Knowledge Sharing and Reuse". *The handbook on Applied Expert Systems*, Liebowitz J. (Ed.), CRC Press.

[Gómez-Pérez&Benjamins99] Gómez-Perez, A. & Benjamins, R. 1999. "Overview of Knowledge Sharing and Reuse Components: Ontologies and Problem-Solving Methods". Procs. of the IJCAI'99 workshop on Ontologies and Problem-Solving Methods (KRR5), Stockholm, Sweden.

[González-Calero97] González-Calero, P.A., 1997. *Aplicación de técnicas basadas en conocimiento como soporte a la reutilización en bibliotecas orientadas a objetos*. Tesis Doctoral, Dpto. Informática y Automática, Universidad Complutense de Madrid.

[González *et al.* 99a] González-Calero, P.A., Gómez-Albarran, M., & Díaz-Agudo, B., 1999. "A substitution-based adaptation model". *Challenges for Case-Based Reasoning* - Proc. of the ICCBR'99 Workshops, Centre for Learning Systems and Applications - Dept. of Computer Science - University of Kaiserslautern (Informe Técnico LSA-99-03E), pp. III-18-III-23.

[González *et al.* 99b] González-Calero, P.A., Gómez-Albarran, M., & Díaz-Agudo, B., 1999. "Applying DLs for Retrieval in Case-Based Reasoning". Proc. of the 1999 International Description Logics Workshop (DL'99), Linköping, Sweden. CEUR Vol. 22.

[González&Dankel97] González, A., & Dankel, D., 1997. *The engineering of knowledge-based systems: theory and practice*. Prentice Hall, Inc.

[Griffith&Domeshek96] Griffith, A., & Domeshek, E., 1996. "Indexing Evaluations of Buildings to Aid Conceptual Design". *Case-Based Reasoning: experiences, lessons, and future directions*. AAAI Press / The MIT Press, Menlo Park, CA., Leake, D. (Ed.), pp. 67-80.

[Grosso *et al.* 98] Grosso, W.E., Gennari, J. H., Fergerson, R.W., Musen, M.A., 1998. "When Knowledge Models Collide (How it Happens and What to Do)", 11th Workshop on Knowledge Acquisition, Modelling and Management (KAW98). Canada.

[Gruber93] Gruber, T.R., 1993. "A translation approach to portable ontologies". *Knowledge Acquisition*, 5(2), pp. 199-220.

[Gruber94] Gruber, T. R., 1994. "Towards Principles for the Design of Ontologies Used for Knowledge sharing". Guarino, N. & Poli, R., (Eds.), Formal Ontology in Conceptual Analysis and Knowledge Representation. Kluwer. También Technical Report KSL-93-04. Knowledge Systems Laboratory, Stanford University, CA.

[Gruber95] Gruber, T. R., 1995. "Towards principles of the design of ontologies used for knowledge sharing". *International Journal of Human-Computer Studies*, 43, pp. 907-928.

[Grüninger&Fox95] Grüninger, M., & Fox, M.S., 1995. "Methodology for the Design and Evaluation of Ontologies", Proxs. Of the IJCAI'95 Workshop on Basic Ontological issues in Knowledge Sharing, Menlo Park, USA. AAAI Press.

[Guarino95] Guarino, N. "Formal Ontology, Conceptual Analysis and Knowledge Representation". *International Journal of Human-Computer Studies*, 43 (5/6), pp.625-640.

[Guarino&Giaretta95] Guarino, N. & Giaretta, P., 1995. "Ontologies and Knowledge bases: Towards a Terminological Clarification". N.Mars (Ed.) Towards very large Knowledge Bases: Knowledge Building and Knowledge Sharing 1995. IOS Press, Ámsterdam, pp. 25-32.

[Guarino97] Guarino, N., 1997. "Understanding, Building and Using Ontologies". A Commentary to "Using Explicit Ontologies in KBS Development", by van Heijst, Schreiber, and Wielinga. *International Journal of Human and Computer Studies*, 46 (2-3), pp. 293-310.

[Hammond89] Hammond, K.J., 1989: Case-Based Planning: Viewing Planning as a Memory Task. Academic Press.

[Heinsohn *et al.* 94] Heinsohn, J., Kudenko, D., Nebel, B., & Profitlich, H., 1994. "An empirical analysis of terminological representation systems". *Artificial Intelligence*,68 (2), pp. 367-398

[Hinrichs92] Hinrichs, T.R., 1992. *Problem Solving in open worlds*. Lawrence Erlbaum Associates.

[Horrocks98] Horrocks, I., 1998. "Using an expressive description logic: FaCT or fiction?". Procs. of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), pp. 636-647.

[Horrocks02] Horrocks, I., 2002. "DAML+OIL: A Description Logic for the Semantic Web", *Bulleting of the IEEE Computer Society Technical Commitee on Data Engineering*, 25 (1), pp. 4-9.

[Horrocks *et al.* 00] Horrocks, I., Fensel, D., Harmelen, F., Decker, S., Erdmann, M., Klein, M., 2000. "OIL in a nutshell". Procs. of the ECAI'00 Workshop on Application of Ontologies and PSMs. Berlin, Germany.

[Horrocks&Patel-Schneider01] Horrocks, I., & Patel-Schneider, P.F., 2001. "The Generation of DAML+OIL". Working Notes of the 2001 International Description Logics Workshop (DL'01) Stanford, USA, CEUR Vol 49.

[Kamp96] Kamp, G., 1996. "Using Description Logics for Knowledge Intensive Case-Based Reasoning". *Advances in Case-Based Reasoning* (EWCBR'96), Smith I. & Faltings B. (Eds.). Springer-Verlag LNAI 1168, pp. 204-218.

[Kamp97] Kamp, G., 1997. "On the admissibility of concrete domains for CBR based on Description Logics", Case-Based Reasoning Research and Development, Procs. of the second International Conference on Case-Based Reasoning (ICCBR'97)., Leake, D.B., Plaza, E. (Eds.), Springer-Verlag, pp. 223-231,.

[Karp *et al.* 99] Karp, R., Chaudhri, V., THomere, J., 1999. "XOL: a XML-Based Ontology Exchange Language". http://www.ai.sri.com/~pkarp/xol

[Kass *et al.* 86] Kass, A. M., Leake, D. B., & Owens, C., 1986. "SWALE: A Program that Explains". *Explanation Patterns: Understanding Mechanically and Creatively*, R. C. Schank (Ed.). Hillsdale, NJ: Lawrence Erlbaum, pp. 232-254.

[Kass90] Kass, A., 1990. *Developing creative hypotheses by adapting explanations*, Ph.D. Thesis, Northwestern University, The Institute for the Learning Sciences, Evanston, IL.

[Kifer *et al.* 95] Kifer, M., Lausen, G., & Wu, J., 1995. "Logical foundations of Object Oriented and Frame-Based Languages". *Journal of the ACM*, Vol. 42

[Kim01] Kim, H.M., 2001. "Predicting How Ontologies for the Semantic Web Will Evolve" (2001). *Communications of the ACM*, submitted.

[Koehler94] Koehler, J., 1994. "An Application of Terminological Logics to Case-based Reasoning". *Procs.of the 4th International Conf. on Principles of Knowledge Representation and Reasoning*, pp. 351-362.

[Koehler96] Koehler, J., 1996. "Planning from Second Principles". *Artificial Intelligence*, Vol. 87, pp. 145-186.

[Kolodner83] Kolodner, J., 1983. "Maintaining organization in a dynamic long-term memory". *Cognitive Science*, 7, pp. 243-280-

[Kolodner93] Kolodner, J., 1993. *Case-based Reasoning*, Morgan Kaufmann, San Mateo CA.

[Kolodner&Leake96] Kolodner, J., Leake, D., 1996. "A Tutorial Introduction to Case-Based Reasoning". *Case-Based Reasoning: experiences, lessons, and future directions*, Leake D.B (Ed.)., AAAI Press / The MIT Press, pp. 31-65.

[Kolodner96] Kolodner, J., 1996. "Making the Implicit Explicit: Clarifying the Principles of Case-Based Reasoning". *Case-Based Reasoning: experiences, lessons, and future directions*, Leake D.B (Ed.)., AAAI Press / The MIT Press, pp. 349-370.

[Koton89] Koton., P., 1989. *Using experience in learning and problem solving.* Massachusetts Institute of Technology, Laboratory of Computer Science, Ph.D. Thesis MIT/LCS/TR-441.

[Lassila&Swick99] Lassila, O., & Swick, R., 1999. "Resource Description Framework (RDF) Model and Syntax Specification". W3C Recommendation. http://www.w3.org/TR/PR-rdf-syntax.html

[Leake92] Leake, D., 1992. *Evaluating Explanations: A Content Theory*. Hillsdale, NJ: Lawrence Erlbaum Associates

[Leake93] Leake, D., 1993**. "**Learning Adaptation Strategies by Introspective Reasoning about Memory Search".*Procs AAAI-93 Case-Based Reasoning Workshop*, volume WS-93-01, Leake, D., (Ed.), pp. 57-63

[Leake *et al.* 95] Leake, D.B., Kinley, A., & Wilson, D., 1995. "Learning to Improve Case Adaptation by Introspective Reasoning and CBR". Procs of the First International Conference on Case-Based Reasoning (ICCBR'95).

[Leake95a] Leake, D., 1995. "Becoming An Expert Case-Based Reasoner: Learning To Adapt Prior Cases". *Proceedings of the Eigth Annual Florida Artificial Intelligence Research Symposium*, pages 112-116.

[Leake95b] Leake, D., 1995. "Representing Self-Knowledge for Introspection about Memory Search". Invited position paper, Procs of the 1995 AAAI Spring Symposium on Representing Mental States and Mechanisms.

[Leake96] Leake, D. (Ed.) ,1996. *Case-Based Reasoning: experiences, lessons, and future directions*. AAAI Press / The MIT Press, Menlo Park, CA.

[Leake *et al.* 96] Leake, D., Kinley, A., & Wilson, D., 1996. "Acquiring Case Adaptation Knowledge: A Hybrid Approach". *Procs of the 13th National Conference on Artificial Intelligence*, AAAI Press, Menlo Park, CA.

[Leake *et al.* 97a] Leake, D., Kinley, A., & Wilson, D., 1997. "A Case Study of Case-Based CBR", ", Case-Based Reasoning Research and Development, Procs. of the second International Conference on Case-Based Reasoning (ICCBR'97)., Leake, D.B., Plaza, E. (Eds.), Springer-Verlag.

[Leake *et al.* 97b] Leake, D., Kinley, A., & Wilson, D., 1997. "Case-Based Similarity Assessment: Estimating Adaptability from Experience". *Procs of the 14th National Conference on Artificial Intelligence*, AAAI Press, Menlo Park, CA.

[Leake *et al.* 01] Leake, D., Smyth, B., Wilson, D., Yang, Q. (Eds.), 2001. *Computational Intelligence Journal*, 17 (2). Special Issue on Case-based Maintenance. Blackwell Publishers, Boston MA UK.

[Lehmann92] Lehmann, F. (Ed.), 1992. *Semantic Networks in Artificial Intelligence.* Int. Series in Modern Applied Math. and Comp. Science, Vol.24. Oxford: Pergamon.

[Lenat&Guha90] Lenat, D.B. & Guha, R.V., 1990. "Building large knowledge-based systems. Representation and inference in the Cyc Project". Addison-Wesley, Massachusetts.

[Levesque&Brachman85] Levesque, H.J., & Brachman, R.J., 1985. "A fundamental tradeoff in knowledge representation and reasoning (revised version)". *Readings in Knowledge Representation*, Brachman, R.J. & Levesque, H.J. (Eds.). Morgan Kaufmann, Los Altos, California, pp. 41-70.

[Lieber&Napoli98] Lieber J., & Napoli, A., 1998. "Correct and Complete Retrieval for Case-Based Problem-Solving", *Procs. of the 13th European Conference on Artificial Intelligence (ECAI'98)*, Prade, H. (Ed.), John Wiley & Sons, ltd. Brighton, UK, pp. 68-72.

[Lopez&Plaza93] Lopez, B., & Plaza, E., 1993. "Case-Base Planning for medical diagnosis". *Methodologies for Intelligent Systems, 7th. International Symposium, ISMIS-93*. Springer Verlag, LNAI 689.

[Lopez&Plaza97] López de Mántaras, R., & Plaza, E., 1997. "Case-Based Reasonig : An Overview". *AI Communications*, 10 (1), IOS Press, pp. 21-29.

[Luke&Heflin00] Luke, S., & Heflin, J., 2000. "SHOE 1.01. Proposed Specification". http://www.cs.umd.edu/project/plus/SHOE/spec1.01.htm

[Mahesh96] Mahesh, K., 1996. "Ontology development for machine translation: ideology and methodology". *Technical Report*, Computer Research Laboratory, New Mexico State University.

[MacGregor&Bates87] Mac Gregor, R., & Bates, R., 1987. "The Loom Knowledge Representation Language". *ISI Reprint Series, ISI/RS-87-188*, University of Southern California.

[MacGregor88] Mac Gregor, R., 1988. "A Deductive Pattern Matcher". *Proc. of the 1988 National Conference on Artificial Intelligence*, St Paul, MN, Ago. 1988.

[MacGregor91] Mac Gregor, R., 1991. "The Evolving Technology of Classification-Based Knowledge Representation Systems". *Principles of Semantic Networks*. Sowa, J. F.(Ed.), Morgan Kaufmann, pp. 385-400.

[Marcus88] Marcus, S., (Ed.), 1988. *Automating knowledge acquisition for expert systems*. Boston, Kluwer.

[Mars92] Mars, N.J., (Ed.), 1992. ECAI'92 Workshop on Knowledge Sharing and Reuse: Ways and Means. European coordinating Committee for Artificial Intelligence (ECCAI), Austria.

[Martín89] Martín, C., 1989, "Indexing using complex features", Procs. of the Case Based Reasoning Workshop, Morgan Kaufmann Publishers, Florida, pp. 26-30.

[Matuscheck&Jantke97] Matuschek, D. & Jantke, K.P., 1997."Axiomatic Characterization of Structural Similarity for Case-Based Reasoning", *Procs of Florida AI Research Symposium*, D.D. Dankel (Ed.), Florida, AI Research Society, pp. 432-436.

[McGuiness *et al.* 00a] McGuinness, D. L., Fikes, R., Rice, J. & Wilder, S., 2000. "An Environment for Merging and Testing Large Ontologies." *Procs of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR'00)*. Breckenridge, Colorado, USA.

[McGuiness *et al.* 00b] McGuinness, D. L., Fikes, R., Rice, J. & Wilder, S., 2000. "The Chimaera Ontology Environment." *Procs of the Seventeenth National Conference on Artificial Intelligence (AAAI'00)*. Austin, Texas.

[Mena *et al.* 00] Mena, E., Illarramendi, A., Kashyap, V., & Sheth, A., 2000. "*OBSERVER: An Approach for Query Processing in Global Information Systems based on Interoperation across Pre-existing Ontologies*", International journal on Distributed And Parallel Databases (DAPD), ISSN 0926-8782, 8(2):223-272.

[Miller90] Miller, G.A., 1990. "WORDNET: an online lexical database". International Journal of Lexicography, 3(4), pp. 235-312.

[Mizoguchi *et al.* 95] Mizoguchi, R., Vanwelkenhuysen, J., Ikeda, M., 1995. "Task Ontology for Reuse of Problem Solving Knowledge". *Towards Very Large Knowledge Bases: Knowledge Building & Knowledge Sharing*. IOS Press, Amsterdam, pp.46-59.

[Motta&Zdrahal96] Motta, E. & Zdrahal, Z., 1996. "Parametric design problem solving", Procs of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop, Alberta, Canada. Gaines, B. R. & Musen, M. A., (Eds.) SRDG Publications, University of Calgary, pp. 9.1-9.20

[Motta98] Motta, E., 1998. *Reusable Components for Knowledge Modeling*. PhD thesis, The Open University, Milton Keynes, United Kingdom.

[Muñoz&Hüllen95] Muñoz-Ávila, H., & Hüllen, J., 1995. ``Retrieving Cases in Structured Domains by Using Goal Dependencies". in *CBR Research and development, Procs of the International Conference on CBR* (ICCBR'95), Springer, pp. 241-252.

[Muñoz&Hüllen96] Muñoz-Ávila, H., & Hüllen, J., 1996. "Feature Weighting by Explaining Case-based Planning Episodes". *Advances in Case-Based Reasoning* (EWCBR'96), Smith, I. & Faltings, B. (Eds.). Springer-Verlag LNAI 1168, pp. 280-294.

[Muñoz *et al.* 99] Muñoz-Avila, H., Aha D., & Breslow L., 1999. "HICAP: An Interactive Case-Based Planning Architecture and its Application to Noncombatant Evacuation Operations",Orlando, FL: AAAI Press, pp. 870-875.

[Musen89] Musen, M.A., 1989. "Automated Support for building and extending expert models". *Machine Learning*, 4, pp. 347-376.

[Napoli *et al.* 96] Napoli, A., Lieber, J., Courien, R., 1996. "Classification-Based Problem Solving in Case-Based Reasoning", *Advances in Case-Based Reasoning* (EWCBR'96), Smith, I. & Faltings; B., (Eds.), Springer-Verlag LNAI 1168, pp. 295-307.

[Napoli&Lieber96] Napoli, A., & Lieber, J., 1996. "Using Classification in Case-Based Planning", *Procs of the 12th European Conference on Artificial Intelligence (ECAI'96),* Wahlster W. (Ed.), John Wiley & Sons, ltd., Budapest, Hungary, pp. 132-136.

[Napoli *et al.* 97] Napoli, A., Lieber, J., & Simon A., 1997. "A Classification-Based Approach to Case-Based Reasoning", Procs of the International Workshop on Description Logics (DL'97), Brachman R., Donini F., Franconi E., Horrocks I., Levy A., Rousset M-C., (Eds.) Université Paris-Sud, Centre d'Orsay, Laboratoire de Recherche en Informatique LRI, URA-CNRS 410.

[Nebel90] Nebel, B., 1990. "Terminological Reasoning is Inherently Intractable". *Artificial Intelligence*, 43, pp. 235-249.

[Nebel&Von Luck88] Nebel, B., & von Luck, K., 1988. "Hybrid reasoning in BACK." Methodologies for Intelligent Systems, Ras, Z. W. & Saitta, L. (Eds) North Holland, Amsterdam, Netherlands, pp. 260-269.

[Neches *et al.* 91] Neches, R., Fikes, R.E., Finin, T.,. Gruber, T.R, Senator, T. & Swartout, W.R., 1991. "Enabling technology for knowledge sharing". *AI Magazine*, 12 (3), pp. 36-56.

[Newel82] Newell, A., 1982. "The Knowledge Level". *Artificial Intelligence*, 18 (1).

[Osborne&Bridge97] Osborne, H. & Bridge, D.G.: "Similarity metrics: a formal unification of Cardinal and not cardinal similarity measures", Case-Based Reasoning Research and Development, Procs. of the second International Conference on Case-Based Reasoning (ICCBR'97)., Leake, D.B., Plaza, E. (Eds.), Springer-Verlag, pp. 235-244.

[Park *et al.* 98] Park, J.Y. Gennari, J.H., & Musen, M.A, 1998. "*Mappings for Reuse in Knowledge-based Systems*". 11 th Workshop on Knowledge Acquisition, Modelling and Management (KAW98). Banff, Canada,.

[Paterson *et al.* 01] Patterson, D., Aamodt, A., & Smyth, B., (Eds.). ICCBR-01 Workshop On Case-Based Reasoning Authoring Support Tools. Technical

Reports of the Navy Center for Applied Research in Artificial Intelligence of the Naval Research Laboratory.

[Patil *et al.* 92] Patil, R. S., Fikes, R. E., Patel-Schneider, P. F., Mckay, D., Finin, T., Gruber, T. R. & Neches, R., 1992. The DARPA Knowledge Sharing Effort: Progress Report. Morgan Kaufmann, Cambridge, MA.

[Pinto *et al.* 99] Pinto, H.S., Gómez Pérez, A., & Martins, J.P., 2000. "Some Issues on Ontology Integration". Procs of the IJCAI'99 Workshop on Ontologies and Problem-Solving Methods:Lessons Learned and Future Trends.

[Plaza&Arcos94] Plaza, E., & Arcos, J.-L., 1994. "Flexible Integration of Multiple Learning Methods into a Problem Solving Architecture". European Conference on Machine Learning (ECML'94), Springer LNCS 784, pp. 403-406

[Plaza95] Plaza, E., 1995. "Cases as Terms: A feature term approach to the structured representation of cases". *Case Based Reasoning Research and development* (ICCBR'95). Veloso, M., & Aamodt, A., (Eds.), Springer LNAI 1010, pp. 265-276

[Plaza&Arcos00] Plaza, E. & Arcos, J.-L., 2000. "Towards a Software Architecture for Case-based Reasoning Systems". *Foundations of Intelligent Systems, 12th International Symposium* (ISMIS'00)**.** Ras, Z. W. & Ohsuga, S., (Eds.), LNCS 1932.

[Porter89] Porter, B.W., 1989. Similarity Assessment: computation vs. representation. Procs of the Workshop on case-based reasoning *(DARPA)*, Morgan Kaufmann, Florida, pp. 203-207, pp- 82-84.

[Prediger&Stumme99] Prediger, S., & Stumme, G., 1999. "Theory driven logical scaling". Proc. of the 1999 International Description Logics Workshop (DL'99), Linköping, Sweden. CEUR Vol. 22.

[Puerta *et al.* 92] Puerta, A.R., Egar, J.W., Tu, S.W. & Musen, M.A., 1992. "A multiple-method knowledge acquisition shell for the automatic generation of knowledge acquisiton tools", *Knowledge Acquisition*, 4 (2), pp. 171-196.

[Puppe93] Puppe, F., 1993. *Systematic Introduction to Expert Systems: Knowledge Representation and Problem Solving Methods*. Springer-Verlag.

[Quilis 85] Quilis, A., 1985. *Métrica Española*, Ariel, Barcelona.

[Rich&Knight91] Rich, E., & Knight, K., 1991. *Artificial Intelligence*. McGraw-Hill, New York, NY, 2nd edition.

[Richter95] Richter, M.M., 1995. "The Knowledge Contained in Similarity Measures", invited talk given at ICCBR'95 in Sesimbra, Portugal, http://www.cbr-web.org/documents/Richtericcbr95remarks.html

[Riesbeck&Schank89] Riesbeck, C.K., & Schank, R.C., 1989. *Inside Case Based Reasoning*. Lawrence Erlbaum Associates, Cambridge.

[Rissland83] Rissland, E., 1983. "Examples in legan reasoning: Legal hypotheticals". *Procs. of the 8th Joint Conference on Artificial Intelligence, IJCAI'83*.

[Rumbaugh *et al.* 98] Rumbaugh, J., Jacobson, I., & Booch, G., 1998. "The Unified Modeling Language Reference Manual". Addison Wesley Co, Reading, Massachussetts.

[Salotti&Ventos98] Salotti, S., & Ventos, V., 1998. "Study and Formalization of a Case-Based Reasoning System using a Description Logic". *Advances in Case-Based Reasoning. Procs of the* 4th *European Workshop on* CBR (*EWCBR'98)*, Smyth B. & Cunningham P. (Eds.), Springer-Verlag (LNAI 1488).

[Salton&McGill 83] Salton, G., & McGill, M.J., 1983. *Introduction to Modern Information Retrieval*. McGraw-Hill.

[Sánchez-León93] Sánchez León, F., 1993. Corpus Resources And Terminology ExtRaction project (MLAP-93/20), 'Spanish tagset for the CRATER project', ftp://ftp.lllf.uam.es/pub/CRATER/esT-tagger-1-0.tar.gz

[Schank&Abelson77] Schank, R., & Abelson, R., 1977. "Scripts, Plans, Goals and Understanding". Hillsdale, NJ: Lawrence Erlbaum Associates.

[Schank82] Schank, R., 1982. *Dynamic Memory*. Cambridge University Press.

[Schreiber *et al.* 94] Schreiber, Th., Wielinga, B., Akkermans, J., Van de Velde, W. & de Hoog, R., 1994. "CommonKADS: A comprehensive methodology for KBS development". *IEEE Expert*, 9(6), December.

[Schreiber *et al.* 00] Schreiber, Th., Akkermans, J., Anjewierden, A., de Hoog, R., Shadbolt, N., Van de Velde, W., & Wielinga, B., 2000. *Knowledge Engineering and Management: The CommonKADS Methodology*, MIT Press.

[Seta *et al.* 99] Seta, K., Ikeda, M., Shima, T., Kakusho, O., Mizoguchi, R., 1999. "CLEPE: a task ontology based conceptual level programming environment". *Trans. of IEICE*, J81-D-II (9), pp.2168-2180.

[Simpson85] Simpson, R., 1985. "A computer model of case-based reasoning in problem solving: An investigation in the domain of dispute mediation". *Technical Report GIT-ICS-85/18*, Georgia Institute of Technology.

[Smyth&Cunningham93] Smyth, B., & Cunningham, P., 1993. "Complexity of Adaptation in Real-World Case-Based Reasoning Systems". Artificial Intelligence & Cognitive Science VI Belfast: Queens University Press.

[Smyth&Keane93] Smyth B. & Keane M.T., 1993. "Retrieving Adaptable Cases" *Topics in Case-based reasoning* (EWCBR'93). Wess, S., Althoff, K.-D., & Richter, M., (Eds.). Springer Verlag, LNAI 837, pp. 106-118.

[Smyth&Keane95] Smyth, B. & Keane, M., 1995. "Remembering to Forget: A Competence-Preserving Deletion Policy for Case-Based Reasoning". Procs of the *14th* International Joint Conference on Artificial Inteligence (IJCAI'95), Montreal, Canada. Morgan Kaufmann, pp. 377-382.

[Smyth&Keane98] Smyth, B. & Keane, M.T., 1998. "Adaptation guided retrieval: questioning the similarity asumption in Reasoning" *Artificial Intelligence* 102, pp. 249-293.

[Smyth99] Smyth, B., 1999. "Constructing Competent Case-Based Reasoners: Theories, Tools and Techniques". *Procs of the Workshop on Automating the Construction of Case-Based Reasoners*. IJCAI-99, Stockholm, Sweeden.

[Smyth&McKenna99] Smyth, B. & McKenna, E., 1999. "Building Compact Competent Case-bases". *Case-Based Reasoning Research and Development* (ICCBR'99), Althoff, K.D., Bergmann, R., Branting, L.K., (Eds.), Springer, LNAI 1650.

[Sowa91] Sowa, J. (Ed.), 1991. *Principles of Semantic Networks*. Morgan Kaufmann Publishers, Inc., San Mateo, California.

[Sowa00] Sowa, J. (Ed.), 2000. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Pacific Grove, CA: Brook/Cole, a division of Thomson Learning.

[Speel&Aben97] Speel, P-H, & Aben., M., 1997. Applying a library of problem solving methods on a real-life task. *International Journal of Human Computer Studies*, 46 (May), pp. 627-652.

[Spibey91] Spibey, P., 1991. *Express language reference manual*. Technical Report ISO 10300 / TC184/SC4/WG 5, CADDETC.

[Steels90] Steels, L., 1990. "Components of expertise". *AI Magazine*, 11(2), pp. 30-49.

[Studer *et al.* 98] Studer, R., Benjamins, V.R., & Fensel, D, 1998. "Knowledge Engineering, principles and methods". *Data and Knowledge Engineering*, 25 (1-2), pp. 161-197.

[Stumme&Maedche01] Stumme, G., & Maedche, A., 2001. "FCA-MERGE: Bottom-Up Merging of Ontologies", Procs. of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01). Nebel, B., (Ed.), Morgan Kaufmann, pp. 225-230

[Swartout *et al.* 97] Swartout, W., Patil, R., Knight, K., & Russ, T., 1997. "Towards distributed use of large-scale ontologies". Spring Symposium Series on Ontological Engineering, pages 33-40, Stanford, 1997. AAAI Press.

[Sycara88] Sycara, K., 1988. "Using case-based reasoning for plan adaptation and repair". Procs. Case-Based Reasoning Workshop, DARPA. Florida. Morgan Kaufmann, pp. 425-434.

[Sycara&Navichandra89] Sycara, K., & Navichandra, D., 1989. "Index transformation and generation for case retrieval", Procs of the Workshop on case-based reasoning *(DARPA)*, Morgan Kaufmann, Florida, pp. 324-328.

[Ullman97] Ullman, J.D., 1997. "Information Integration using logical views". Procs of ICDT'97, LNCS 1186, pp. 19-40, Springer.

[Uschold96] Uschold, M., 1996. "Building ontologies: Towards a unified methodology", 16th Annual Conf. of the British Computer Society Specialist Group on Expert Systems

[Uschold&Gruninger96] Uschold, M. & Gruninger, M., 1996. Ontologies: Principles, methods and applications. *Knowledge Engineering Review*, 11(2).

[Uschold *et al.* 98] Uschold, M., Clark, P., Healy, M., Williamson, K., & Woods, S., 1998. "Ontology Reuse and Applications". *Procs of the International Conference on Formal Ontology and Information Systems - FOIS'98 (Frontiers in AI and Applications v46)*, Guarino, N. (Ed.), Amsterdam:IOS Press, pp. 179-192,

[Valente *et al.* 98] Valente, A., Breuker, J., & Vand de Velde, W., 1998. "The CommonKADS Library in Perspective". *International Journal of Human-Computer Studies*, 49 (4), pp. 391-416.

[Van Harmelen&Fensel99] Van Harmelen, F., & Fensel, D., 1999."Surveying notations for machine-processable semantics of Web sources". Procs. of the IJCAI'99 Workshop on Ontologies and PSMs.

[Van Heijst *et al.* 97] Van Heijst, G., Schreiber, A.T., and Wielinga, B.J., 1997. "Using Explicit Ontologies in Knowledge Based Systems Development", *International Journal of Human and Computer Studies*, 46 (2/3), pp. 183-292.

[Veloso94] Veloso, M., 1994. "Planning and learning by analogical reasoning", LNAI 886 Springer.

[Veloso *et al.* 96] Veloso, M., Munoz-Avila, H., & Bergmann R., 1996 General-purpose case-based planning: Methods and systems, *AI Communications*, 9 (3), pp. 128-137.

[Ventos *et al.* 98] Ventos, V., Brézellec, P., Soldano, H., & Bouthinon, D., 1998. "Learning Concepts in C-CLASSIC(delta/epsilon)", *Procs of the 1998 International Workshop on Description Logics (DL'98)*. Franconi E., De Giacomo G., MacGregor R.M., Nutt W., Welty C.A., (Eds) Trento, Italy, pp. 50-54

[Vossen99] Vossen, P., 1999. *EuroWordnet General Document*. Version 3, Final. http://www.hum.uva.nl/ewn

[Watson97] Watson, I., 1997. *Applying Case-Based Reasoning: Techniques for Enterprise Systems*. Morgan Kaufman Publishers.

[Wielinga&Breuker86] Wielinga, B.J., & Breuker, J.A., 1986. "Models of Expertise". Procs. of the European Conference on Artificial Intelligence (ECAI'86), pp. 306-318.

[Wielinga *et al.* 92] Wielinga, B.J., Schreiber, A.T., & Breuker, J.A., 1992. "KADS: A modelling approach to knowledge engineering". *Knowledge Acquisition*, 4 (1), pp. 5-53, Special Issue "The KADS approach to knowledge engineering". Reprinted in: Buchana, B and Wilkins, D (Eds.), 1992, Readings in Knowledge Acquisition and Learning, Morgan Kaufmann, pp. 92-116.

[Wille92] Wille, R., 1992. "Conceptual Lattices and conceptual knowledge systems". *Computers and Mathematics with Apps*, 23 (6-9), pp. 493-515.

[Woods91] Woods, W. A., 1991. "Understanding Subsumption and Taxonomy: A Framework for Progress". *Principles of Semantic Networks*. Sowa, J. F.(Ed.), Morgan Kaufmann, pp. 45-94.

APÉNDICE A. LOOM

Como hemos descrito en el Capítulo 3, los distintos sistemas de DLs difieren entre sí en una serie de características. El objetivo deseado por todos es conseguir una alta expresividad, un tiempo de respuesta lo suficientemente rápido como para poder ser utilizados en aplicaciones interactivas y la completitud de sus razonamientos. El sistema LOOM ofrece una gran expresividad en sus lenguajes de representación de conocimiento (terminológico y asertivo) a costa de sacrificar algo de la completitud de sus procesos de razonamiento. Este apéndice describe la sintaxis y la semántica de los lenguajes de descripción de términos y aserciones de LOOM (extraídas de [Brill93]). Así mismo, presentaremos el lenguaje de construcción de consultas que acompaña al sistema y las dos características que afectan al tipo de inferencias realizables, el razonamiento hacia adelante y la semántica de mundo abierto, así como las medidas adoptadas para reducir el coste de las estrategias de razonamiento hacia adelante.

1. Lenguajes terminológicos de LOOM

1.1 El lenguaje de definición de conceptos

```
concept-expr ::=
            ConceptName |
            ( {:AND | :OR} concept-expr+ ) |
            ( :ONE-OF {Number+ | InstanceId+} ) |
            ( :THROUGH Number Number ) |
            ( {:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation-expr ) |
            ( {:ALL | :SOME | :THE} relation-expr ConceptName ) |
            ( {:FILLED-BY | :NOT-FILLED-BY} relation-expr
                                    {InstanceId | Constant}+ ) |
            ( {:SAME-AS | :SUBSET} relation-expr relation-expr) |
            ( {<  | > | ≤ | ≥ | = | <>} relation-expr
                                    {relation-expr | Number} ) |
            ( :RELATES relation-expr relation-expr
                                    {relation-expr | Constant} ) |
            (:SATISFIES  ( ?Var ) query-expr ) |
            set-expr ;
```

Observamos que la definición de un concepto puede incluir:

- Otros conceptos definidos previamente lo que significa que cualquier instancia del primero deberá ser también instancia del segundo, heredando además todas sus restricciones. Se distinguen dos conceptos especiales: THING, concepto predefinido que encabeza la jerarquía de conceptos, e INCOHERENT, un concepto predefinido, sin instancias y que se encuentra situado en la parte más inferior de la jerarquía.
- Restricciones acerca de la cardinalidad de las relaciones que se aplican a las instancias del concepto que está siendo definido. Las limitaciones pueden referirse a un número mínimo de valores de la relación (`at-least`), un número máximo (`at-most`) o un número concreto (`exactly`).
- Restricciones sobre el tipo de los valores de las relaciones aplicadas sobre las instancias del concepto. Se puede restringir el tipo de todos los valores (`all`, para relaciones multivaluadas, o `the`, para relaciones univaluadas) o el de algunos valores (`some`).
- Restricciones sobre los valores concretos de las relaciones aplicadas sobre las instancias del concepto que se define. Se puede indicar que el valor válido es uno de

un cierto grupo (`one-of`), el/los valores concretos que siempre deben figurar (`filled-by`) o el/los valores que no pueden aparecer (`not-filled-by`).

- Relaciones entre los valores de dos relaciones diferentes. Puede que los valores de dos relaciones sean los mismos (`same-as`) o que los de una relación constituyan un subconjunto de los valores de otra (`subset`).
- La conjunción o la disyunción de cualquiera de los operadores anteriores.
- El constructor satisfies permite añadir fórmulas con variables al lenguaje de construcción de conceptos utilizando una expresión válida del lenguaje de construcción de consultas de LOOM.

Además de la sintaxis del lenguaje, queremos dar aquí una idea intuitiva de la semántica de sus operadores de construcción. La Tabla A-1 incluye la semántica denotacional de los operadores de construcción de conceptos de LOOM. Suponemos que el concepto que estamos construyendo es C. Su definición puede incluir:

- C= D, donde A es el nombre de otro concepto (`ConceptName`). Significa que cualquier instancia de C debe ser también una instancia de A. Por ejemplo, la restricción "una persona es un mamífero" indica que cualquier instancia del concepto *persona* debe ser una instancia del concepto *mamífero*. Establece de forma explícita el enlace *C es-un A*.
- C=(**:and** A_1 .. A_n) significa que cualquier instancia del concepto C es también instancia de cada uno de los conceptos A_1 .. A_n. El concepto C es una especialización de todos los conceptos A_1 .. A_n y, además, generaliza a cualquier otro concepto que especialice a todos los conceptos A_1 .. A_n. Por ejemplo, si definimos que un *trabajador* es una persona *adulta*, entonces cualquier instancia del concepto *trabajador* debe ser también instancia de los conceptos *persona* y *adulto*.
- C=(**:or** A_1 .. A_n) significa que cualquier instancia del concepto C es también instancia de al menos uno de los conceptos A_1 .. A_n. El concepto C es una generalización de todos los conceptos A_1 .. A_n y, además, especializa a cualquier otro concepto que generalice a todos los conceptos A_1 .. A_n. Por ejemplo, si definimos el concepto *persona* como *hombre* o *mujer*, cualquier instancia de *persona* es instancia del concepto *hombre* o del concepto *mujer*. En este ejemplo los conceptos son disjuntos, pero en general esto no tiene porqué ocurrir.
- C=(**:all** R A) significa que cualquier instancia i de C cumple que sólo se relaciona a través de la relación R con instancias del concepto A ($\forall$R. A). Por ejemplo, la restricción "*la nacionalidad de una persona es un país*" indica que todas las instancias de la relación *nacionalidad* que tengan una instancia del concepto *persona* como primer elemento tienen una instancia del concepto *país* como segundo elemento. Una instancia de una relación es una dupla ordenada de individuos conectados por dicha relación. Si "*la nacionalidad de Luis es España*" entonces <Luis, España> es una instancia de la relación *nacionalidad*.
- C=(**:some** R A) significa que cualquier instancia de C se relaciona a través de la relación R con algún individuo que es instancia del concepto A ($\exists$R.A). Por ejemplo, si definimos que alguno de los *componentes* de un *coche* son *ruedas*, entonces cualquier instancia del concepto *coche* tiene algún componente que es una instancia del concepto *rueda*.
- C=(**:the** R A) significa que cualquier instancia de C se relaciona a través de la relación R con (exactamente) un individuo que además es instancia del concepto A

THING	Δ^I (todos los individuos del dominio)
INCOHERENT	$\varnothing$ (la semántica es la de un concepto sin instancias)
A	A^I
(**:and** A_1 .. A_n)	$A_1{}^I \cap .. \cap A_n{}^I$
(**:or** A_1 .. A_n)	$A_1{}^I \cup .. \cup A_n{}^I$
(**:all** R A)	$\{d \in \Delta^I \mid R^I(d) \subseteq A^I\}$
(**:some** R A)	$\{d \in \Delta^I \mid R^I(d) \cap A^I \neq \varnothing\}$
(**:the** R A)	$\{d \in \Delta^I \mid \lvert R^I(d)\rvert = 1, R^I(d) \in A^I\}$
(**:at-least** n R)	$\{d \in \Delta^I \mid \lvert R^I(d)\rvert \geq n\}$
(**:at-most** n R)	$\{d \in \Delta^I \mid \lvert R^I(d)\rvert \leq n\}$
(**:exactly** n R)	$\{d \in \Delta^I \mid \lvert R^I(d)\rvert = n\}$
(**:same-as** R S)	$\{d \in \Delta^I \mid R^I(d) = S^I(d)\}$
(**:subset** R S)	$\{d \in \Delta^I \mid R^I(d) \subseteq S^I(d)\}$
(**:filled-by** R b_1...b_m)	$\{d \in \Delta^I \mid b_1{}^I \in R^I(d), ...,b_m{}^I \in R^I(d)\}$
(**:filled-by** R b)	$\{d \in \Delta^I \mid b^I \in R^I(d)\}$
(**:not-filled-by** R b_1...b_m)	$\{d \in \Delta^I \mid b_1{}^I \notin R^I(d), ...,b_m{}^I \notin R^I(d)\}$
(**:one-of** b_1...b_m)	$\{ b_1{}^I ..... b_m{}^I \}$
(**:through** n_1 n_2)	$\{ n_1{}^I, (n_1+1)^I ..... n_2{}^I \}$
(**<** R S)	$\{d \in \Delta^I \mid R^I(d) < S^I(d)\}$,
(**:relates** R R_1 R_2)	$\{d \in \Delta^I \mid \forall c: c \in R_1{}^I(d): \forall b: b \in R_2{}^I(d): (c,b) \in R^I\}$

Notación:

Se introduce una notación funcional para las relaciones de forma que $e \in R^I(d) \Leftrightarrow (d,e) \in R^I$

$\lvert C \rvert$ es la cardinalidad del conjunto C.

Tabla A-1. Semántica denotacional de los constructores de conceptos de LOOM

$(\geq 1.R \cap \leq 1.R \cap \forall R.\ A)$. Por ejemplo, si definimos el concepto *europeos* como los individuos cuya *nacionalidad* es instancia del concepto *país europeo*, y Luis es una instancia de *europeos*, entonces se relacionará mediante la relación *nacionalidad* con una instancia de *país europeo*.

- C=(**:at-least** n R) significa que cualquier instancia de C se relaciona a través de la relación R por lo menos con *n* individuos ($\geq$ n.R). Por ejemplo, la restricción "*un padre tiene al menos un hijo*" indica que todas las instancias del concepto *padre* deben formar parte de al menos *una* instancia de la relación *hijo (at-least 1 hijo)*.
- C=(**:at-most** n R) significa que cualquier instancia de C se relaciona a través de la relación R como mucho con *n* individuos ($\leq$ n.R). Por ejemplo, la restricción "*una persona tiene como mucho un padre*", indica que cualquier instancia del concepto *persona* se relaciona a través de la relación *padre*, como mucho con 1 instancia de *persona*.
- C=(**:exactly** n R) significa que cualquier instancia de C se relaciona exactamente con *n* individuos a través de la relación R ($\geq$n.R $\cap$ $\leq$n.R). Por ejemplo, el concepto "*familias con 3 hijos*", representa a todas las familias que tienen exactamente 3 instancias de la relación *hijo*.

- C=(**:same-as** R S) significa que cualquier instancia de C se relaciona con el mismo conjunto de individuos a través de la relación R que de la relación S. Por ejemplo, podemos definir el concepto "*viaje de ida y vuelta*" como un *viaje* donde "*el origen es el mismo que el destino*", siendo *origen* y *destino* sendas relaciones entre *viajes* y *lugares*.
- C=(**:subset** R S) significa que cualquier instancia de C se relaciona a través de la relación R con un subconjunto del conjunto de individuos con que se relaciona a través de la relación S. Por ejemplo, si incluimos en la definición del concepto persona la restricción `(:subset conocidos amigos)` indica que cualquier instancia de *persona* conoce a todos sus amigos.
- C=(**:filled-by** R b_1...b_n) significa que cualquier instancia de C se relaciona a través de la relación R con b_1, ..,y con b_n. Por ejemplo, la restricción "*la nacionalidad de un español es España*" se podría incluir en la definición del concepto *español*, donde *España* es una instancia del concepto *país* `(:filled-by nacionalidad España)`.
- C= (**:not-filled-by** R b_1...b_n) significa que cualquier instancia de C no se relaciona ni con b_1, .. ni con b_n a través de la relación R. Por ejemplo podemos definir el concepto *no español* como `(:not-filled-by nacionalidad España)`.
- C=(**:one-of** b_1...b_m) significa que el concepto C se define por enumeración de sus instancias, es decir, por extensión. Cualquier instancia de C será b_1.o ... o b_m. Por ejemplo podemos definir el concepto *estaciones* como `(:one-of Otoño Primavera Verano Invierno).`
- C=(**:through** n_1 n_2) significa que el concepto C es el intervalo numérico entre n_1 y n_2 (que son instancias del concepto predefinido `Number`) Por ejemplo podemos definir el intervalo valido para los *días* como `(:throught 1 31)`.
- C=(**<** R S) significa que cualquier instancia I de C se relaciona a través de la relación R con valores numéricamente menores que a través de S. El rango de las relaciones R y S debe ser numérico. Los otros operadores aritméticos `>, <=, >=, = y <>` tienen significados análogos. Por ejemplo si definimos dos relaciones *número_de_amigos* y *número_de_conocidos* (de rango numérico) en el concepto personas podemos incluir la restricción `(<= número_de_amigos número_de_conocidos).`
- C=(**:relates** R R_1 R_2) significa que Para cualquier instancia I de C se cumple que si I se relaciona con I_1 a través de R_1 y con I_2 a través de R_2, I_1 e I_2 se relacionan mediante R. Por ejemplo, el concepto que representa a todos los *hombres* cuyas *mujeres* son las *madres* de sus *hijos*, se escribe como `(relates madre esposa hijo)` que quiere decir: "*si María es la mujer de Juan, y Pepe es el hijo de Juan, entonces María es la madre de Pepe*".
- C=(**:satisfies** (?X) Q) significa que Para cualquier instancia I de C, se cumple que la consulta Q de LOOM se satisface cuando la variable ?X está ligada a I.

1.2 El lenguaje de definición de relaciones

```
relation-expr ::=
            RelationName |
            ( :AND relation-expr+ ) |
            ( {:DOMAIN | :RANGE} concept-expr ) |
            ( :DOMAINS concept-expr concept-expr+ ) |
            ( :INVERSE relation-expr ) |
            ( :COMPOSE relation-expr+ ) |
            ( :SATISFIES ( ?Var ?Var+ ) query-expr );
```

S	S^I
BINARY TUPLE	$\Delta^I \times \Delta^I$
(**:and** S_1 .. S_n)	$S_1{}^I \cap \ldots \cap S_n{}^I$
(**:domain** C)	$\{(d,d') \in (\Delta^I \times \Delta^I) \mid d \in C^I\}$
(**:range** C)	$\{(d,d') \in (\Delta^I \times \Delta^I) \mid d' \in C^I\}$
(**:domains** C_1 ... C_{n-1})	$\{(d_1,..,d_n) \in (\Delta^I \times ..\times \Delta^I) \mid d_1 \in C_1{}^I, ..., d_{n-1} \in C_{n-1}{}^I$
(**:inverse** S)	$\{(d,d') \in R^I \mid (d',d) \in S^I\}$ R es la relación que estamos definiendo.
(**:compose** R_1 .. R_n)	$R_1{}^I \circ \ldots \circ R_n{}^I$ siendo la operación ($\circ$), {(d1,d2)} $\circ$ {(d2,d3)} = {(d1,d3)}

Tabla A-2. Semántica denotacional de los constructores de relación de LOOM

Observamos que definición de una relación puede incluir:

- Otra relación definida con anterioridad.
- Restricciones sobre el tipo del dominio (`domain`) o del rango (`range`) de la relación.
- Una relación puede ser el resultado de componer (`compose`) otras existentes.
- Una relación se puede definir como la inversa (`inverse`) de otra definida previamente.
- La conjunción de cualquiera de las anteriores.

Además de la propia definición, a los conceptos y a las relaciones se les pueden asociar otros atributos. Algunas de las restricciones involucran a un único concepto mientras que otras involucran a más de uno.

Incluimos una descripción informal de los operadores de construcción del lenguaje de definición de relaciones. En la Tabla A-2 se incluye la semántica denotacional de los operadores. Recordamos que una instancia de una relación R es una tupla ordenada de individuos relacionados mediante R. Suponemos que la relación que estamos construyendo es R., la definición puede incluir:

- R= S, significa que R es igual que otra relación definida previamente S. Por ejemplo, podemos definir la relación *pariente* basándonos en la relación *hijo*.
- R=BINARY-TUPLE significa que R es igual que la relación que ocupa la posición más arriba de la jerarquía de relaciones binarias, es decir subsume a cualquier otra relación binaria.
- R= (:**and** S_1 .. S_n) significa que cualquier instancia de la relación R, también es instancia de las relaciones S_1 ,.., S_n. Por ejemplo, podemos definir la relación "muy amigo" como "pariente y amigo", de forma que si "Luis es muy amigo de Juan" entonces "Luis es amigo de Juan" y "Luis es pariente de Juan".
- R= (:**domain** C) significa que si $<i_1,i_2>$ es una instancia de R, entonces i_1 es instancia del concepto C. Por ejemplo, se puede incluir en la definición de la relación *hijo* que los elementos de su dominio son instancias del concepto *persona*.

- R= (:**domains** C_1 ..C_{n-1}) significa que si <i_1,...,i_n> es una instancia de R, entonces $\forall j.(1..n-1)$ se cumple que i_j es instancia del concepto C_j. Por ejemplo, se puede definir la relación *hijo* como una relación de aridad 3 en la que los elementos de sus dominios son instancias de los conceptos *hombre* y *mujer*.
- R= (:**range** C) significa que si <i_1,i_2> es una instancia de R, entonces i_2 es instancia del concepto C. Por ejemplo, se puede incluir en la definición de la relación *hijo* que los elementos de su rango deben ser instancias del concepto *persona*.
- R= (:**inverse** S) significa que si <i_1,i_2> es una instancia de R, entonces <i_2,i_1> es una instancia de S. Por ejemplo, la relación *padre* se puede definir como la inversa de la relación *hijo*, de forma que cuando se identifique una instancia <*p, h*> de la relación *hijo*, automáticamente se detecte que la dupla <*h*, *p*> es una instancia de la relación *padre*.
- R=(:**compose** R_1 .. R_{n-1}) significa que para cualquier instancia <i_1,...,i_n> de R se cumple que <i_i, i_{i+1}> es instancia de R_i para cualquier i entre 1 y n-1. Por ejemplo, podemos definir la relación *cuñado* como la composición de las relaciones *esposa* y *hermano*, de forma que si <p_1, p_2> es una instancia de la relación *esposa* (p_2 es la esposa de p_1) y <p_2, p_3> es una instancia de la relación *hermano*, entonces se infiere que <p_1, p_3> es una instancia de la relación *cuñado*
- R=(**:satisfies** ($?X_1$ .. $?X_n$) Q) significa que para cualquier instancia <i_1..i_n>de R se satisface la consulta Q cuando las variables $?X_1$ .. $?X_n$ se ligan (en orden) a i_1..i_n.

2. Los axiomas terminológicos y otras restricciones adicionales

En LOOM, al igual que en otros sistemas de DLs, además de la propia definición, podemos asociar a los conceptos y relaciones otros atributos que restringen el conjunto de interpretaciones a ser consideradas. Esto incluye, por ejemplo, la definición de términos *primitivos* o *definidos* y la creación de *particiones* o particiones *exhaustivas* sobre un concepto. En este apartado vamos a describir algunas de estas restricciones, que se añaden a la definición de los términos (conceptos y relaciones) como argumentos adicionales.

Utilizando **is/is-primitive** se puede definir un término (tanto los conceptos como las relaciones) como definido o como primitivo. Como ya vimos, los *términos definidos* son aquellos en los que la descripción recoge el conjunto de condiciones *necesarias* y *suficientes* que deben verificar las instancias de ese término, mientras que en los *términos primitivos* las restricciones que aparecen en la descripción son tan sólo las condiciones *necesarias* que se aplican a las instancias de ese término. En caso de no utilizar ninguno de los dos, por defecto el término que se define se considera una especialización primitiva de THING.

Los conceptos definidos reconocen como instancias suyas aquellos individuos que satisfacen su descripción. Las relaciones definidas permiten identificar como instancias de dicha relación a las tuplas ordenadas de individuos que satisfagan su definición.

Entre las restricciones añadidas que involucran a un único concepto/relación nos encontramos con:

- Los argumentos **implies** y **defaults** que se utilizan para asociar una regla a un concepto. La definición es, como antes señalábamos, el conjunto de condiciones necesarias y suficientes para identificar que un individuo es una instancia del término definido (en el caso de los términos primitivos tan sólo representan condiciones

necesarias). Mediante los argumentos *implies y defaults*, se puede asociar, además, con la definición de un concepto o relación, un conjunto de condiciones necesarias, que no se consideran como intrínsecas de la entidad representada, y por lo tanto no se incluyen en su definición, pero que, una vez se ha identificado que un individuo es instancia del concepto o de la relación, se aplicarán sobre él. Las reglas introducidas con *implies* son estrictas, es decir se aplican siempre que un individuo se clasifique como instancia del término. Las implicaciones introducidas con *default* sólo se aplican si son consistentes con el estado actual de la base de conocimiento, es decir, se aplicarán siempre que no entren en contradicción con la información asertada explícitamente.

- Mediante el argumento **characteristics** podemos añadir características que condicionan el tipo de razonamiento que se puede realizar sobre la entidad que está siendo definida. Por ejemplo, los conceptos o relaciones se pueden marcar como **closed-world**, indicando así que sobre ellos se aplicará una semántica de mundo cerrado, en lugar de la semántica de mundo abierto que se aplica por defecto. También se puede indicar, por ejemplo, que una relación debe ser necesariamente univaluada con **single-valued**.
- Mediante los argumentos **predicate** y **function** podemos definir conceptos y relaciones utilizando procedimientos o funciones, que escapan a los mecanismos de razonamiento y clasificación de la lógica descriptiva, pero que llegan donde los operadores del lenguaje terminológico no pueden llegar. Así, con un concepto se puede asociar una función implementada en un lenguaje de propósito general (Lisp en el caso de LOOM) que, al ser aplicada sobre un individuo, determine si el individuo es una instancia del concepto, o bien una función que al ser invocada devuelva las instancias del concepto. Y una relación se puede definir por medio de una función que, a partir del primer elemento de la dupla, obtenga el segundo.

Además de las restricciones asociadas con conceptos individuales, es posible definir restricciones que involucran a más de un concepto. En concreto, utilizando los argumentos **partitions** y **exhaustive-partitions** es posible representar que las instancias de un concepto deben estar organizadas en un cierto número de clases disjuntas. Por ejemplo, podemos representar que los conceptos *mamífero* y *ave* constituyen una partición del concepto *vertebrado*. Los conceptos *mamífero* y *ave* son disjuntos y subconceptos de *vertebrado* y cumplen que no puede existir ningún individuo que sea al mismo tiempo instancia de *mamífero* y de *ave*, aunque sí puede haber instancias del concepto *vertebrado* que no sea instancia ni de *mamífero* ni de *ave*. Si la partición es *exhaustiva*, todas las instancias del concepto deben ser instancias de una, y sólo una, de las clases que componen la partición. Así, por ejemplo, los conceptos *mamífero*, *ave*, *reptil* y *pez* constituyen una partición exhaustiva del concepto *vertebrado*. Utilizando el argumento *in-partition* indicamos que un concepto forma parte de una partición. Si un concepto forma parte de una partición (por ejemplo *mamífero* es parte de la partición de los *vertebrados*) debe especializar al concepto sobre el que el estamos construyendo la partición, es decir el concepto *mamífero* debe especializar al concepto *vertebrado*.

La Tabla A-3 muestra la semántica para los axiomas terminológicos de LOOM, donde podemos sustituir C por cualquiera de las construcciones de la Tabla A-1 y R por las construcciones de la Tabla A-2.

(**defconcept** A)	$A^I \subseteq \Delta^I$
(**defconcept** A **:is** C)	$A^I \equiv C^I$
(**defconcept** A **:is-primitive** C)	$A^I \subseteq C^I$
(**defrelation** P **:is** R)	$P^I = R^I$
(**defrelation** P **:is-primitive** R)	$P^I \subseteq R^I$
(**defrelation** P)	$P^I \subseteq \Delta^I \times \Delta^I$
(**disjoint** A_1 ... A_n)	$A_i^I \cap A_j^I = \varnothing$, $i \neq j$
(**partition** B A_1 ... A_n)	$\cup_{k=1..n} A_k^I \subseteq B^I$, $A_i^I \cap A_j^I = \varnothing$, $i \neq j$
(**exhaustive-partition** B A_1 ... A_n)	$\cup_{k=1..n} A_k^I = B^I$, $A_i^I \cap A_j^I = \varnothing$, $i \neq j$
(**implies** A B)	$A^I \subseteq B^I$

Tabla A-3. Semántica denotacional de los axiomas terminológicos y restricciones adicionales del lenguaje LOOM

3. Lenguaje asertivo de LOOM

Las aserciones que es posible construir en LOOM son las obtenidas con el lenguaje asertivo --o de definición de individuos—siguiente:

```
proposition ::=
            ( concept instance ) |
            ( relation instance+ value ) |
            ( :CREATE ?Var concept [:CLOS] ) |
            ( :SAME-AS instance instance ) |
            ( :ABOUT instance about-clause* )
about-clause ::=
            concept | ( concept ) | ( relation value ) |
            (:FILLED-BY relation value+) | (:FILLED-BY-LIST relation list ) |
            ( {:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation ) |
            ( {:ALL | :SOME | :THE} relation concept )
```

Las restricciones indican que:

- Se puede asertar que un individuo es instancia de un concepto previamente definido.
- Se puede asertar que un individuo está relacionado con otro a través de una cierta relación.
- Se pueden imponer restricciones sobre la cardinalidad o el tipo de los valores de las relaciones aplicables sobre un individuo.

En LOOM se pueden eliminar aserciones y en consecuencia el sistema cuenta con un mecanismo de mantenimiento de la verdad. El lenguaje que se emplea para eliminar aserciones es el mismo que el que se utiliza para construirlas (utilizado en una expresión `forget` en vez de en una expresión `tell`).

4. Lenguaje de consultas de LOOM

Además de los dos lenguajes anteriores, terminológico y asertivo, LOOM cuenta con un lenguaje para recuperar individuos de la base de conocimiento que cumplan una serie de restricciones expresables a través de dicho lenguaje. Las consultas que se pueden realizar pueden ser muy elaboradas, incluyendo descripciones parciales de los datos a recuperar Incluimos la sintaxis de este lenguaje, que cubre un subconjunto de la lógica de predicados de primer orden:

```
query-expr ::=
     ( {:AND | :OR} query-expr+) |
     ( {:NOT | :FAIL} query-expr ) |
     ( :IMPLIES query-expr query-expr ) |
     ( {:FOR-SOME | :FOR-ALL} ( ?Var+) query-expr ) |
     ( :COLLECT ( ?Var ) query-expr ) |
     ( concept instance ) |
     ( relation instance+ value ) |
     ( :SAME-AS instance instance ) |
     ( :SUBSET instance instance ) |
     ( :PREDCALL LispPredicate value+ ) |
     ( :ABOUT instance about-clause* )
```

5. Razonamiento hacia adelante y semántica de mundo abierto

Las dos características que, por defecto, afectan al tipo de inferencias que se realizan en LOOM son el hecho de que tanto el clasificador como el reconocedor "razonen hacia adelante" (*forward chaining*) [MacGregor91] y el que se suponga una semántica de "mundo abierto" que es común a los sistemas de DLs y a otros sistemas de representación de conocimiento (aunque no en sistemas de bases de datos). Esta característica normalmente se utilizan en contextos en los que no se puede asumir que el conocimiento sea completo [Brachman *et al.* 91].

Cuando un sistema razona hacia adelante se infiere todo lo que es posible inferir a partir de la información de que se dispone. Esta es lo que hace que, cada vez que se introduce una nueva instancia en la ABox, o se modifica alguna existente, se revisen automáticamente los cambios que ello pueda producir en los enlaces *instancia-de* correspondientes a los tipos y que han sido calculados previamente, ya afecten los cambios al propio individuo o a otros con los que se encuentre relacionado.

El principal inconveniente que tiene el uso de una estrategia de razonamiento hacia adelante es su elevado coste. En LOOM se introduce una serie de restricciones y facilidades que permiten reducir dicho coste. Podemos resumirlas en las dos siguientes:

- El clasificador no utiliza la información disponible acerca de los individuos. Como hemos visto, la definición de conceptos puede hacer referencia a individuos (a través de restricciones como `filled-by` o `one-of`). Sin embargo la información acerca de dichos individuos no se utiliza para el cálculo de la relación de subsunción y, en consecuencia, tampoco se utiliza en el proceso de clasificación. De esta forma, la inclusión de una nueva aserción en la base de conocimiento no puede provocar una reclasificación de la jerarquía (jerarquía que, por otra parte, se considera esencialmente estática). Si este tipo de reclasificaciones se admitiesen, podrían dar

lugar a nuevas inferencias en la ABox, lo que a su vez provocaría una nueva reclasificación, y así sucesivamente.

- Los conceptos se pueden marcar de modo que sobre ellos se realice razonamiento "hacia atrás". Por defecto los conceptos están marcados como "hacia adelante", esto es, el reconocedor se encarga en todo momento de mantener actualizada la lista de individuos y sus tipos a partir de las aserciones conocidas. Sin embargo, cuando un concepto se marca como "hacia atrás" el reconocedor sólo calcula la relación *instancia-de* bajo demanda explícita, reduciéndose así la carga computacional. Loom utiliza la estrategia de implementación del reconocedor que se basa en la relación "es instancia de".

En los sistemas en los que, además de la realización de aserciones y del almacenamiento de los resultados del reconocimiento de instancias, se permite la eliminación de asertos en la ABox, el uso de razonamiento hacia adelante requiere la inclusión de algún mecanismo de mantenimiento de la verdad que preserve la corrección de los enlaces *instancia-de* almacenados. Dicho mecanismo debe asegurar en todo momento que (a) todos los hechos que es posible inferir a partir de los hechos asertados se han inferido, y (b) cada hecho inferido tiene su prueba lógica o demostración asociada. Así las cosas, cada vez que un nuevo hecho se aserta o se elimina, el mecanismo de mantenimiento de la verdad deberá revisar el conjunto de hechos inferidos.

El segundo aspecto influyente en las inferencias posibles es la suposición de una semántica de "mundo abierto", esto es, cualquier hecho que no sea conocido por el sistema (cualquier hecho no asertado) no se supone que es falso, simplemente se considera desconocido y, como tal, incapaz de provocar inferencias.

Por ejemplo, supongamos que los únicos hechos conocidos sobre Ana son: "*Ana es una persona*" `Persona(Ana)` y "*Luis es su hijo*" `tiene_hijo(Ana, Luis)`. En una base de datos con semántica de mundo cerrado se interpretaría que `Luis` es el *único* hijo de `Ana`. Sin embargo en una ABox con semántica de mundo abierto se interpreta únicamente que `Luis` es hijo de `Ana` y que no sabemos si `Ana` tiene o no más hijos. Por tanto, aunque se sepa que `Luis` es una instancia del concepto `Hombre` (consultar la terminología de la Figura 3-X) no se podría deducir que todos los hijos de `Ana` son hombres. Para expresar que `Luis` es el único hijo de `Ana` habría que hacer el aserto explícito de que `Ana` tiene un hijo: `(tell (:about Ana (:exactly 1 tiene_hijo)))` o también `(tell (:about Ana (:at-most 1 tiene_hijo)))`

Esta característica viene impuesta, en la práctica, por el hecho de que se razone hacia adelante. Si se considerara una semántica de "mundo cerrado", lo que no se aserta se considera falso y, por lo tanto, también da lugar a inferencias. Dado que, gracias al razonamiento hacia adelante, cualquier cambio en la base de conocimiento provoca la realización de todas las inferencias posibles, encontraríamos situaciones en las que los hechos que se asertan con posterioridad invalidan inferencias realizadas en el pasado, aumentando así de manera intolerable la carga del sistema de mantenimiento de la verdad.

5.1 Ejemplo de la incompletitud de LOOM

Como vimos en el Capítulo 3, en LOOM se sacrifica la completitud del razonamiento para poder así proporcionar un lenguaje terminológico bastante expresivo y una implementación eficiente de los mecanismos de razonamiento. Esta característica hace de LOOM un sistema de representación apto para ser utilizado en aplicaciones de tamaño y complejidad reales. El

problema de esta decisión es que en LOOM no están implementadas todas las inferencias que la semántica de los constructores permite. Sirva como muestra el siguiente ejemplo:

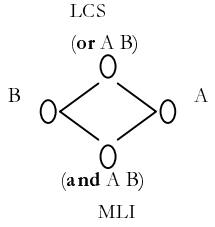
1. Creamos una partición exhaustiva sobre el concepto a:
   ```
   (defconcept a :exhaustive-partitions $a$)
   ```
2. Creamos los conceptos *c* y *d* que forman una particion exhaustiva del concepto *a*, de forma que cualquier instancia de *a* debe ser instancia de *c* o *d*, pero en ningun caso podrá ser instancia de ambos.
   ```
   (defconcept c   :is-primitive a :in-partition $a$)
   (defconcept d   :is-primitive a :in-partition $a$)
   ```
3. Creamos la relación *rel* `(defrelation rel)`
4. Creamos una subrelación *subrel1* de la relación *rel*. Aquellas instancias de *rel* cuyo rango sea instancia de *c* se reconocen automáticamente como instancias de *subrel1*
   ```
   (defrelation subrel1 :is (:and rel (:range c)))
   ```
5. Creamos otra subrelación *subrel2* de la relación *rel*. Aquellas instancias de *rel* cuyo rango sea instancia de *d* se reconocen automáticamente como instancias de *subrel1*
   ```
   (defrelation subrel2 :is (:and rel (:range d)))
   ```
6. Creamos un concepto e tal que:
   - Cualquier instancia *i* de *e* se relaciona a través de *rel* sólo con instancias del concepto *a*.
   - Como mínimo se relaciona con 3 individuos mediante *rel*.
   - Como máximo se relaciona con 1 individuo mediante *subrel1*.
   - Como máximo se relaciona con 1 individuo mediante *subrel2*.

   ```
   (defconcept e
          :is (:and (:all rel a)
                    (:at-least 3 rel)
                    (:at-most 1 subrel1)
                    (:at-most 1 subrel2)))
   ```

Aunque el sistema debería detectar que el concepto *e* es incoherente no lo hace. La causa de la incoherencia radica en que las anteriores definiciones implican que el concepto *e* debe verificar `(:and (:at-most 2 rel) (:at-least 3 rel))` lo cual es evidentemente imposible.

5.2 Least Common Subsummer

La mayoría de los sistemas de DLs incluyen constructores similares a THING que es el concepto que subsume a todos los demás, y INCOHERENT, que es el concepto que es subsumido por todos los demás. También existe un constructor de conceptos de tipo AND, cuya semántica es la intersección de conjuntos. Estos constructores convierten el espacio (infinito) de las descripciones en una estructura matemática denominada "semirretículo de intersección" (*meet semi-lattice*), donde para cualquier par de descripciones *A* y *B* se puede *encontrar* (se puede *construir*, ya que no todos los conceptos del semirretículo están creados) el *máximo límite inferior* –una descripción que generaliza a cualquier otra descripción que especialice a *A* y *B*– como **and**(A, B).



Por ejemplo, dados los conceptos:

```
(defconcept C5
    :is (:and CURSO
          (:at-most 20 alumno)
          (:all profesor (:one-of Gauss Euclides))))
(defconcept C6
    :is (:and CURSO DIVERTIDO
          (:at-most 25 alumno)
          (:all profesor (:one-of Gauss Marx))))
```

El **máximo límite inferior** (MLI) (*Greatest Common Subsumee* GCS) será:

```
:is (:and CURSO DIVERTIDO
          (:all profesor (:one-of Gauss))
          (:at-most 1 profesor)
          (:at-most 20 alumno))
```

En LOOM lo podemos obtener utilizando la función `compute-conjunction-concept` que crea y clasifica un concepto que es la conjunción de los términos dados (conceptos o relaciones), que deben estar previamente creados y clasificados. El nuevo concepto creado tiene un nombre asignado por el sistema:

```
(compute-conjunction-concept (list (fc c5) (fc c6)))  ⇒ |C|c5&c6
```

También lo podríamos obtener mediante el constructor de conceptos `and` como:

```
(defconcept MLI :is (:and c5 c6))
```

El **mínimo límite superior** (LCS) *Least-common-subsummer)* será el concepto con la siguiente definición:

```
:is (:and CURSO
           (:all profesor (:one-of Gauss Euclides Marx))
           (:at-most 3 profesor)
           (:at-most 25 alumno))
```

que no se puede obtener con ninguna función de LOOM aunque sí se puede definir el concepto `(:or C5 C6)` que será una generalización de C5 y de C6. El concepto anterior se obtiene como: `(defconcept MLS :is (:or c5 c6))`

Otra opción es utilizar la función `find-subsumers&subsumes` que devuelve una lista de los conceptos más específicos por encima, por debajo y equivalentes a cualquier descripción de concepto dada, teniendo en cuenta únicamente los conceptos de la base de conocimiento actual (no crea ninguno adicional). El LCS de un conjunto de conceptos se obtiene como el primer valor que devuelve la función aplicada a la expresión formada por el OR de los conceptos en cuestión. Es decir, `(find-subsumers&subsumees '(:or A B))` ➔ `LCS (A,B) GCS (A,B) equivalent-concept`

El tercer valor devuelto el nombre de un concepto equivalente de la base de conocimiento, si hay alguno, o NIL si el concepto no existe (no está definido).

APÉNDICE B: FORMALIZACIÓN DE CBRONTO EN LOOM

1. Terminología general

```
;; Raices de las jerarquias de terminos de CBROnto.
   (defconcept CBROnto-concept
       :annotations ((documentation "the root for all the CBROnto concepts.")))
   (defrelation CBROnto-relation
    :annotations ((documentation "the root for all the CBROnto relations.")))
   ;; raiz de los conceptos del dominio.
   (defconcept domain-concept
      :annotations ((documentation "the root for all the domain entities.")))
   ;;;;; raiz de las relaciones del dominio.
   (defrelation domain-relation
    :annotations ((documentation "The root of the domain relation hierarchy.")))
```

1.1 Lenguaje de representación de casos

```
   (defconcept case
       :is-primitive (:and CBROnto-concept
       (:all has-case-component case-component)
                  (:the has-description case-description)
                  (:all has-solution case-solution) (:at-most 1 has-solution)
```

```
                (:all has-result case-result))
      :annotations ((CurrentMeasure Default_Measure)
                    (documentation "the class of the cases.")))
   (defconcept case-component :is-primitive CBROnto-concept)
   (defconcept case-description :is-primitive case-component
       (:all kind-of-reasoning reasoning-type))
   (defconcept reasoning-type :is-primitive (:and CBROnto-Concept (:one-of
   diagnosis evaluate explain design resolve search))

   (defconcept case-solution :is-primitive case-component)
   ;; Hay dos tipos de soluciones transformacionales y derivacionales.
   (defconcept Transformational_Solution :is case-solution)
   (defconcept Derivational_Solution
       :is (:and case-solution
                 (:all has-sequence-step Solution-Step)
                 (:at-least 1 has-sequence-step)))

   (defconcept Solution-step :is-primitive CBROnto-concept
     (:the do-action Action)
     (:at-most 1 over) (:all over Domain-Concept)
     (:all actor Domain-Concept)
     (:the sequence-number Integer))

   (defconcept case_with_solution
       :is (:and case  (:at-least 1 has-solution)))
   (defconcept Goal :is-primitive CBROnto-concept)
   (defconcept Action :is-primitive CBROnto-concept)
   (defconcept Precondition :is-primitive CBROnto-concept)
   (defconcept Properties :is-primitive Precondition)
   (defconcept case-result :is-primitive case-component)
   (defconcept Success-Result :is-primitive case-result)

   (defconcept Failure-Result :is-primitive
       (:and case-result  (:at-most 1 failure-solution)
                          (:all failure-solution Failure-Case-Solution)
                          (:at-most 1 repaired-solution)
                          (:all repaired-solution Case-Solution)
                          (:the repair-with Repair-Strategy)))
   (defconcept Success-result :is-primitive case-component)

   ;; Relaciones
   (defrelation has-description
       :is-primitive (:and has-case-component)
          :annotations ((documentation "Link each case individual
        with a case-description instance representing the
        description of the case.")))
   (defrelation kind-of-reasoning
       :is-primitive (:and CBROnto-relation
                           (:domain case-description)
                           (:range reasoning-type))
   :annotations ((documentation "This relation joins a case with the kind of
reasoning performed by the case (search, explains, design, resolve, ..). ")))
```

```
 (defrelation has-precondition :is-primitive
    (:and (:range Precondition))
  :annotations ((documentation "case preconditions are certain restrictions
that must be satifisfied before applying the case solution.")))
(defrelation gets :is-primitive
   (:and (:range Goal))
   :annotations ((documentation "the gets relationship specify the goals
that are achieved by the case.")))
(defrelation has-solution
   :is-primitive (:and has-case-component (:range case-solution))
  :annotations ((documentation "joins a case with its solution.")))
(defrelation has-sequence-step
    :is-primitive (:and CBROnto-relation (:range Solution-Step)))
(defrelation has-result
    :is-primitive (:and has-case-component (:range case-result))
    :annotations ((documentation "Link each case individual with a case-
description instance representing the  result of the case.")))
(defrelation do-action
   :is-primitive  (:and  (:domain solution-step) (:range action))
   :annotations ((documentation "the perform relationship will
join each solution-step with the action performed by a certain solution
entity of the case.")))
(defrelation sequence-number
   :is-primitive (:and (:domain solution-step) (:range integer))
  :annotations ((documentation "the sequence-number relation links each
solution-step with its order number in the action sequence.")))
(defrelation over
   :is-primitive (:and (:domain solution-step)(:range domain-concept))
  :annotations ((documentation "the over relationship will join each
solution-step with the domain entity that receives the actions of this
solution step.")))
(defrelation actor
   :is-primitive (:and (:domain solution-step)(:range domain-concept))
  :annotations ((documentation "the actor relationship will join each
solution-step with the actor that does the actions of this solution
step.")))
(defrelation failure-solution
    :is-primitive CBROnto-relation :domain failure-result
    :range failure-case-solution)
(defrelation repaired-solution :is-primitive CBROnto-relation
:range Case-solution)
```

1.1.1 Tipos de casos predefinidos

```
(defconcept compound_description
    :is (:and case-description (:at-least 1 composition)))
(defconcept temporal_description
    :is (:and case-description (:at-least 1 temporal)))
(defconcept spatial_description
    :is (:and case-description (:at-least 1 spatial)))
(defconcept design_description
```

```
    :is (:and case-description  (:at-least 1 design-issue)
           (:all space space-desc)(:all system system-desc))
(defconcept descriptive_description
   :is (.and case-description (:at-least 1 description-property)))
(defconcept description_with_goals
    :is (:and Case-Description (:at-least 1 gets) (:all gets Goal))
    :annotations ((documentation "the class of the case descriptions
  having at least one goal.")))
(defconcept description_with_restrictions
   :is (:and Case-Description (:at-least 1 restriction)
   :annotations ((documentation "the class of the case descriptions having
at least one restriction.")))
(defconcept compound_solution
    :is (:and case-solution (:at-least 1 composition))
     :annotations ((documentation "the class of the case-solution
individuals having a compound solution")))
(defconcept temporal_solution
     :is (:and case-solution (:at-least 1 temporal))
     :annotations ((documentation "the class of the case-solution
individuals having a temporal solution.")))
(defconcept spatial_solution
    :is (:and case-solution (:at-least 1 spatial)))
     :annotations ((documentation "the class of the case-solution
individuals having a spatial solution described by means of a spatial
relation.")))
(defconcept solution_with_restrictions
   :is (:and Case-Solution (:at-least 1 restriction)
   :annotations ((documentation "the class of the case solutions having at
least one restriction.")))
(defconcept case_with_compound_description
    :is (:and case (:some has-description compound_description)))
(defconcept case_with_temporal_description
    :is (:and case (:some has-description temporal_description)))
(defconcept case_with_spatial_description
    :is (:and case (:some has-description spatial_description)))
 (defconcept case_with_design_description
     :is (:and case (:some has-description design_description)))
 (defconcept case_with_descriptive_description
    :is (:and case (:some has-description descriptive_description)))
(defconcept case_with_description_with_restrictions
    :is (:and case (:some has-description description_with_restrictions)))
(defconcept Case_with_Goals
    :is (:and Case (:some has-description description_with_goals))
     :annotations ((documentation "the class of the case individuals
described by goals")))
(defconcept Case_with_Restrictions
    :is (:and Case
           (:or   (:some has-description description_with_restrictions)
```

```
                    (:some has-solution solution_with_restrictions))))
   (defconcept Case_with_Derivational_Solution
       :is (:and Case (:some has-solution Derivational_Solution)))
   (defconcept Case_with_Descriptive_Description
      :is (:and Case (:some has-description descriptive_description)))
   (defconcept Case_with_Compound_Description
      :is (:and Case (:some has-description compound_description)))
   (defconcept Case_with_Temporal_Description
      :is (:and Case (:some has-description temporal_description)))
   (defconcept Case_with_Temporal_Solution
      :is (:and Case (:some has-solution temporal_solution)))
   (defconcept Case_with_Architectural_Description
      :is (:and Case (:some has-description design-description)))
   (defconcept Case_with_Architectural_Solution
      :is (:and Case (:some has-solution design-description)))
   ;; el concepto design-description se define en el apartado de diseño
   arquitectónico.
;; Además de la clasificación automática que se hace debajo de CASE según las
características, existen tipos de casos, clasificados bajo el concepto Case-
type. Algunos están predefinidos pero el diseñador puede añadir más.
   (defconcept Case-Type :is-primitive Case)
   (defconcept Descriptive_Case
       :is (:and Case-Type Case_with_Descriptive_Description.
        :annotations ((documentation "los casos descriptivos se utilizan para
describir algo de forma estática en función de sus características descriptivas
observables")))
  (defconcept Compound_Case :is (:and Case-Type Case_with_Compound_Description)
     :annotations ((documentation "Casos estructurales o compuestos")))
   (defconcept Design_Case
       :is (:and Case-Type (:same-as has-description has-solution))
       :annotations ((documentation "Casos de diseño que son los casos en los
que la descripción y la solución coinciden")))
   (defconcept Temporal_Design_Case
       :is (:and Design_Case Case_with_Temporal_Description
              Case_with_Temporal_Solution))
   (defconcept Compound_Design_Case
       :is (:and Design_Case Case_with_Compound_Description
              Case_with_Compound_Solution))
   (defconcept Architectural_Design_Case
       :is (:and Design_Case Case_with_Architectural_Description
              Case_with_Architectural_Solution))
   (defconcept Planning_Case
       :is (:and Case-Type Case_with_Restrictions
              Case_with_Goals Case_with_Derivational_Solution))
```

1.1.1.1. Terminología para describir casos de diseño arquitectónico

```
   (defrelation design-issue
       :domain design-description
       :annotations ((documentation "the goal intended when creating this
   artifact.")))
   (defrelation space
       :domain design-description
```

```
    :range space-desc
    :annotations ((documentation "the spatial description of the
artifact.")))

(defrelation system
    :domain design-description
    :range system-desc
    :annotations ((documentation "the functional description of the
artifact.")))

(defconcept design-description
    :is (:and case-description
         (:at-least 1 design-issue)
         (:all space space-desc)
         (:all system system-desc))
    :annotations
    ((documentation "The design descriptions are divided in two main parts:
spatial description (space) and funcional description (system)")))

(defconcept space-description
  :annotations ((documentation "is a part of the designed artifact that is
described spatially. An space can be composed of other spaces. To describe
an space two kind of attributes are used, one referred to the internal
organization of the composing spaces(i.e. how the composing spaces are
related), and the other one referred to the space description itself.")))
;;;;;;;;;;;;;;;;;;;; Relaciones para describir espacios ;;;;;;;;;;;;;;;
(defrelation space-orientation
    :domain space-desc
    :annotations ((documentation "orientation of the space with respect to
the place where it is positioned.")))

(defrelation space-primary-roles
    :domain space-desc
    :annotations ((documentation "Who is primarily using this space")))

(defrelation space-secondary-roles
    :domain space-desc
    :annotations ((documentation "Other (secondary) users of the space")))

(defrelation space-props
    :domain space-desc
    :annotations ((documentation "static objects asociated with the
space use (as video projector, or black board, or tables, ....")))
(defrelation exterior-openings
    :domain space-desc
    :annotations ((documentation "how many exterior acceses")))

(defrelation exterior-materials
    :domain space-desc
    :annotations ((documentation "Material of the space exterior"))
    )
```

```
(defrelation interior-materials
    :domain space-desc
    :annotations ((documentation "Material of the space interior")))

(defrelation space-form
    :domain space-desc
    :annotations ((documentation "The three dimensional form of the space"))

(defrelation space-function
    :domain space-desc
    :annotations ((documentation "The funcion of the space")))

(defrelation space-characteristics
    :domain space-desc
    :annotations ((documentation "Other characteristics of this space")))

(defrelation included-spaces
    :domain space-desc
    :range space-desc
    :annotations ((documentation "Relates an spaced with other spaces
that are included in it."))          )

(defrelation related-spaces
    :domain space-desc
    :range space-desc
    :annotations ((documentation "Relates an spaced with other spaces
that are related (but not included) with it."))
    )
;;;;;;; las siguientes relaciones son atributos de related-spaces
;;;;;;; y de included-spaces.

(defrelation rel-type
    :annotations ((documentation "represents the kind of relation
between the spaces related by means of related-spaces.
rel-type is intented to be used as an annotation of the relations:
related-spaces and included-spaces.")))

(defrelation rel-strength
    :annotations ((documentation "represents the strength of relation
between the spaces related by means of related-spaces.
rel-type is intented to be used as an annotation of the relations:
related-spaces and included-spaces.")))

(defrelation rel-distance
    :annotations ((documentation "represents the distance about the
 relation between the spaces related by means of related-spaces.
rel-type is intented to be used as an annotation of the relations:
related-spaces and included-spaces.")))
```

```
   (defconcept system-desc
       :annotations ((documentation "is a part of the designed artifact that
   is described by its funcionality")))
```

1.2 Tipos de indices

```
(defconcept IndexType
       :is-primitive CBROnto-Concept (:at-least 1 index-restrictions))
(defrelation index-restrictions
       :is-primitive CBROnto-Relation :domain IndexType)
(tell (:about semantic-type IndexType
       (index-restrictions '((binary-tuple Thing))))
(tell (:about composition-type IndexType
                     (index-restrictions '((composition)))))
(tell (:about spatial-type IndexType (index-restrictions '((spatial)))))
(tell (:about temporal-type IndexType (index-restrictions '((temporal)))))
(tell (:about causal-type IndexType (index-restrictions '((causing)))))
(tell (:about function-type IndexType
       (index-restrictions '((gets Goal)(has-precondition Precondition)))))
(tell (:about description-type IndexType
       (index-restrictions'((description-property)))))
```

1.3 Tipos de usuarios y consultas

```
(defconcept internal-concept
  :is-primitive CBROnto-concept
  :annotations ((documentation " Internal concepts are used as implementation
tools, and  should not be taken into account for other purposes.")))
(defconcept userdef :is-primitive CBROnto-concept
 :annotations ((documentation " User defined is used to mark the user defined
types of cases.")))
(defconcept Query-Type
       :is-primitive (:and Case-Type (:all tosolve_task CBR_Task))
;; El usuario puede definir tipos de consultas como subconceptos de Query-Type
;; y asociarles ciclos CBR de resolución de problemas (es decir instancias de
;; CBR_task.
(defconcept User_Types
       :is-primitive (:and CBROnto_Concept (:all tosolve_task CBR_Task)))
;; User_Types representa los tipos de usuario que define el diseñador,
;; definiendo subconceptos de User_Types. Cada tipo de usuarios finales del
sistema tendrán asignados las instancias de CBR_Task que correspondan al
comportamiento configurado para cada uno de ellos.
(defconcept Designer_User
       :is-primitive (:and CBROnto_Concept (:all tosolve_task CBROnto_Task))
       :annotations ((documentation "tipo de usuario predefinido que describe
al propio diseñador y que tiene asignadas las tareas de diseño")))
(defrelation tosolve_task :is-primitive CBROnto_Relation)
```

1.4 Tipos de relaciones

```
(defrelation documentation :is-primitive (:and CBROnto-relation
                      (:range String)))
```

```
(defrelation importance :is-primitive (:and CBROnto-relation))
(defrelation informative :is-primitive importance
 :annotations ((documentation "informative relations won't be used during
the CBR processes, but are used only for documentation purposes")))
 (defrelation relevant :is-primitive importance)
(defrelation weight :is-primitive (:and CBROnto-relation))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;  relaciones de descripcion ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Relacion generica. Todas las relaciones que se usan para describir
;; algo ser•n subrelaciones suyas. Se usa como raiz del •rbol de relaciones
;; descriptivas.
(defrelation Description-Relation
  :is-primitive  CBROnto-relation
  :annotations ((documentation "The class of all the descriptive relations.")))
;; Restricciones
(defrelation restriction
    :is-primitive Description-Relation
    :annotations ((documentation "Properties that must be safisfied by the case
description to apply the solution")))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Restricciones lite son aquellas que se pueden relajar.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defrelation lite-restriction :is-primitive
  (:and restriction)
  :annotations ((documentation "Properties that shoud  be checked before
applying the solution but can be relaxed. ")))
;;;;;;;;;;;;;;;;;;;; description por propiedades ;;;;;;;;;;;;;;;;;;;;;;;;
(defrelation description-property :is-primitive Description-Relation
  :annotations ((documentation "The properties used to describe something")))

(defrelation has-symptom :is-primitive Description-Relation
     :annotations ((documentation "The symptoms used to describe something"))
       :characteristics (:closed-world))
;;;;;;;;;;;; descripcion por composicion de partes ;;;;;;;;;;;;;;;;;;;;
;; Las relaciones de composici□n se usan cuando se quiere describir algo en
;; base a las partes que lo forman.
(defrelation composition :is-primitive Description-Relation
     :annotations ((documentation "The composition rels are used to
describe something by means of the composing parts.")))
(defrelation part_of :is-primitive composition)
(defrelation has_part :is (:and composition (:inverse part_of)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;; descripcion por organizaci□n espacial ;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defrelation spatial :is-primitive Description-Relation
             :annotations ((documentation "The spatial rels are used to
describe something by means of the spatial layout.")))
(defrelation touching :is-primitive (:and spatial))
(defrelation near_of :is-primitive (:and spatial))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;; descripcion por causas o dependencias ;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defrelation causing :is-primitive Description-Relation
  :annotations ((documentation "The causing rels are used to
describe something by means of the dependency with other elements.")))

(defrelation depends_on :is-primitive (:and causing)
 :annotations ((documentation "i1 depends_on i2 represents a dependency
relationship of i1 respect to i2, i.e. the changes on i2 will affect i1.")))
(defrelation cause :is-primitive (:and causing)
      :annotations ((documentation "No documentation available")))
(defrelation explain :is-primitive (:and causing)
             :annotations ((documentation "No documentation available")))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;; descripcion por organizaci□n temporal ;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defrelation temporal :is-primitive Description-Relation
      :annotations ((documentation "No documentation available")))
(defrelation during :is-primitive temporal
      :annotations ((documentation "No documentation available")))
(defrelation before :is-primitive temporal
      :annotations ((documentation "No documentation available")))
(defrelation after :is (:and temporal     (:inverse before))
      :annotations ((documentation "No documentation available")))
```

1.5 Descriptores del contexto y cualificadores del conocimiento

```
(defconcept Context :is-primitive
(:and CBROnto-Concept
       (:the casebase Case-Type) (:at-most 1 query) (:all Query-Type)
       (:exactly 1 domain-concept-taxonomy-depth)
       (:exactly 1 domain-concept-taxonomy-width)
       (:exactly 1 domain-relation-taxonomy-depth)
       (:exactly 1 domain-relation-taxonomy-width)
       (:exactly 1 case-base-size) (:exactly 1 focus-concept)
       (:exactly 1 focus-concept-taxonomy-depth)
       (:exactly 1 goal-depth) (:exactly 1 precondition-depth)
       (:exactly 1 adaptation-strategy depth)
       (:exactly 1 repair-strategy-depth)
       (:exactly 1 problem-Types_depth) (:exactly 1 fail-Type-depth)
       (:exactly 1 goal-organization-structure)
       (:exactly 1 pre-organization-structure)
       (:exactly 1 case-organization-structure)
       (:exactly 1 case-base-organization-structure)
       (:exactly 1 similarityMeasure_depth)
       (:exactly 1 depends_on_depth)(:exactly 1 depends_on_uses)))

;; individuo que representa el contexto actual.
(tell (:about current-context Context)

;; Atributos que describen el contexto.
(defrelation query :is-primitive CBROnto-relation
:annotations ((documentation "individuo que representa la consulta actual")))
```

```
(defrelation casebase :is-primitive CBROnto-relation
:annotations ((documentation " individuo que representa a la base de casos
sobre la que quiere restringir la aplicación de los métodos (por defecto es
Case con lo que trabajamos con todos los casos)")))

(defrelation domain-concept-taxonomy-width
   :is-primitive CBROnto-relation
   :annotations ((documentation "anchura de la taxonomía de conceptos")))
(defrelation domain-relation-taxonomy-width
   :is-primitive CBROnto-relation
   :annotations ((documentation "anchura de la taxonomía de relaciones")))
(defrelation domain-relation-taxonomy-depth
   :is-primitive CBROnto-relation
  :annotations ((documentation "profundidad de la taxonomía de relaciones")))
(defrelation domain-concept-taxonomy-depth
   :is-primitive CBROnto-relation
   :annotations ((documentation "profundidad de la taxonomía de conceptos")))
(defrelation case-base-size
   :is-primitive CBROnto-relation
   :annotations ((documentation "numero de casos en la base de casos
seleccionada en el atributo casebase del contexto")))
(defrelation focus-concept
   :is-primitive CBROnto-relation
   :annotations ((documentation "individuo que representa el concepto del
dominio bajo y sobre el cual me interesa comprobar la profundidad de las
jerarquías. Se usa para el método de clasificacion. El focus concept es adonde
Lleva la cadena de relaciones. ")))
(defrelation focus-concept-taxonomy-depth
   :is-primitive CBROnto-relation
   :annotations ((documentation "profundidad de la taxonomía del dominio bajo
el concepto representado en el atributo focus-concept")))
(defrelation goal-depth :is-primitive CBROnto-relation
   :annotations ((documentation " profundidad de la jerarquía de objetivos ")))
(defrelation precondition-depth :is-primitive CBROnto-relation
   :annotations ((documentation "profundidad de la jerarquía de
precondiciones")))
(defrelation adaptation-Strategy depth :is-primitive CBROnto-relation
   :annotations ((documentation "profundidad de la jerarquía de estrategias de
adaptación ")))
(defrelation repair-strategy-depth :is-primitive CBROnto-relation
   :annotations ((documentation "profundidad de la jerarquía de estrategias de
reparación ")))
(defrelation Problem-Types_depth :is-primitive CBROnto-relation
   :annotations ((documentation "profundidad de la jerarquía de tipos de
problemas")))
(defrelation fail-Type-depth :is-primitive CBROnto-relation
:annotations ((documentation "profundidad de la jerarquía de tipos de
problemas")))
;; Atributos para determinar las estructuras de organización.
(defconcept structure-type :is-primitive (:and CBROnto-Concept (:one-of
FCA_lattice structure-none)))
(defrelation goal-organization-structure
```

```
:is-primitive CBROnto-relation :range structure-type
:annotations ((documentation "estructura de organización de objetivos")))
(defrelation pre-organization-structure
:is-primitive CBROnto-relation :range structure-type
:annotations ((documentation "estructura de organización de precondiciones")))
(defrelation case-organization-structure
:is-primitive CBROnto-relation :range structure-type
:annotations ((documentation "estructura de organización de casos (bajo
Case)")))
(defrelation case-base-organization-structure
:is-primitive CBROnto-relation :range structure-type
:annotations ((documentation "estructura de organización de casos (bajo la base
de casos seleccionada en el atributo casebase del contexto")))
(defrelation SimilarityMeasure_depth
   :is-primitive CBROnto-relation
   :annotations ((documentation "número de medidas de similitud definidas, es
decir, número de instancias del concepto Similarity_Measure")))
(defrelation Depends_on_depth
   :is-primitive CBROnto-relation
   :annotations ((documentation "número de relaciones clasificadas bajo la
relación depends_on ")))
(defrelation Depends_on_uses
   :is-primitive CBROnto-relation
   :annotations ((documentation "número de veces que se usa cualquier relacion
clasificada como depends on (incluyendo depends_on) en la ABox ")))
;; CONCEPTOS DEFINIDOS QUE REPRESENTAN LOS CUALIFICADORES DE CONOCIMIENTO
(defconcept Knowledge-Qualifiers :is-primitive CBROnto-Concept)
(defconcept With_Query
       :is (:and (:exactly 1 query) (:the query Query-Type))
       :implies Knowledge-Qualifiers)
(defconcept With_CaseBase_with_Description
       :is (:and (:the case-base Case_WithDescription))
       :implies Knowledge-Qualifiers))
(defconcept With_CaseBase_with_Description
       :is (:and (:the case-base Case_WithDescription))
       :implies Knowledge-Qualifiers))
(defconcept With_CaseBase_with_compound_description
       :is (:and (:the case-base case_with_compound_description))
       :implies Knowledge-Qualifiers))
(defconcept With_CaseBase_with_temporal_description
   :is (:and (:the case-base case_with_temporal_description))
       :implies Knowledge-Qualifiers))
(defconcept With_CaseBase_with_spatial_description
   :is (:and (:the case-base case_with_ spatial_description))
       :implies Knowledge-Qualifiers))
(defconcept With_CaseBase_with_design_description
   :is (:and (:the case-base case_with_design_description))
       :implies Knowledge-Qualifiers))
(defconcept With_CaseBase_with_descriptive_description
   :is (:and (:the case-base case_with_ descriptive_description))
       :implies Knowledge-Qualifiers))
(defconcept With_CaseBase_not_Empty
```

```
       :is (:and (> case-base-size 0)) :implies Knowledge-Qualifiers))
(defconcept With_Case_Base_Not_Big
       :is (:and (> case-base-size 0) (< case-base-size 50))
       :implies Knowledge-Qualifiers))
(defconcept With_Case_Base_Small_Size
       :is (:and (> case-base-size 0) (< case-base-size 20))
       :implies Knowledge-Qualifiers))
(defconcept With_Case_Base_Medium_Size
       :is (:and (>= case-base-size 20) (< case-base-size 50))
       :implies Knowledge-Qualifiers))
(defconcept With_Case_Base_Large_Size
       :is (:and (>= case-base-size 50))
       :implies Knowledge-Qualifiers))
(defconcept With_Case_Base_FCA_organized
       :is (:and (:filled-by case-base-organization-structure FCA_lattice))
       :implies Knowledge-Qualifiers))
(defconcept With_Query_with_Goals
       :is (:and With_Query (:the query Query_with_Goals))
       :implies Knowledge-Qualifiers))
(defconcept Query_with_Goals :is (:and Query-Type (:at-least 1 gets)))
(defconcept With_Query_with_Pre
       :is (:and With_Query (:the query With-Pre))
       :implies Knowledge-Qualifiers))
(defconcept Query_with_Pre :is (:and Query-Type (:at-least 1 has-precondition)))
(defconcept With_Domain_Concept_Taxonomy_Shallow_Depth :is
       (< domain-concept-taxonomy-depth 5)
       :implies Knowledge-Qualifiers)))
(defconcept With_Domain_Concept_Taxonomy_Medium_Depth :is
       (:and (>= domain-concept-taxonomy-depth 5)
       (< domain-concept-taxonomy-depth 10))
       :implies Knowledge-Qualifiers)))
(defconcept With_Domain_Concept_Taxonomy_Deep :is
       (>= domain-concept-taxonomy-depth 10)
       :implies Knowledge-Qualifiers)))
(defconcept With_Domain_Concept_Taxonomy_Narrow :is
       (< domain-concept-taxonomy-width 5)
       :implies Knowledge-Qualifiers)))
(defconcept With_Domain_Concept_Taxonomy_Medium_Width :is
       (:and (>= domain-concept-taxonomy-width 5)
              (< domain-concept-taxonomy-width 15))
       :implies Knowledge-Qualifiers)))
(defconcept With_Domain_Concept_Taxonomy_Wide :is
       (> domain-concept-taxonomy-width 15)
       :implies Knowledge-Qualifiers)))
(defconcept With_Goals :is (>= goal-depth 2)
       :implies Knowledge-Qualifiers)))
(defconcept With_Goal_Lattice :is
       (:the goal-organization-structure FCA_lattice)
       :implies Knowledge-Qualifiers)))
(defconcept With_Goal_Taxonomy_Shallow_Depth :is
       (:and (>= goal-depth 2) (< goal-depth 4))
       :implies Knowledge-Qualifiers)))
```

```
(defconcept With_Goal_Taxonomy_Medium_Depth :is
       (:and (>= goal-depth 4) (< goal-depth 8))
       :implies Knowledge-Qualifiers)))
(defconcept With_Goal_Taxonomy_Deep :is
       (:and (>= goal-depth 8))
       :implies Knowledge-Qualifiers)))
(defconcept With_Precondition :is (>= precondition-depth 2)
       :implies Knowledge-Qualifiers)))
(defconcept With_Precondition_Lattice
       :is (:the pre-organization-structure FCA_lattice)
       :implies Knowledge-Qualifiers)))
(defconcept With_Pre_Taxonomy_Shallow_Depth :is
       (:and (>= pre-depth 2) (< pre-depth 4))
       :implies Knowledge-Qualifiers)))
(defconcept With_Pre_Taxonomy_Medium_Depth :is
       (:and (>= pre-depth 4) (< pre-depth 8))
       :implies Knowledge-Qualifiers)))
(defconcept With_Pre_Taxonomy_Deep :is
       (:and (>= pre-depth 8))
       :implies Knowledge-Qualifiers)))
(defconcept With_Similarity_Measures :is
       (:and (>=similarityMeasure_depth 2)) :implies Knowledge-Qualifiers)))
```

Por motivos de espacio no se incluye el resto de los cualificadores que se definen de forma análoga.

1.6 Terminología adicional para los métodos

```
;; Recuperación por Criterios de relevancia
(defrelation relevanceCriteria :is-primitive (:and CBROnto-relation))
;; Criterios predefinidos en forma de relaciones
(defrelation more-on-point
   :constraints (domains Case Case)
   :range Case
   :is (:satisfies (?c1 ?c2 ?cfs)
               (:and   (Case ?c1)(Case ?c2)(Case ?cfs)
                       (neq ?c1 ?cfs)
                       (:for-all ?p (:implies
                                       (:and (Description-Property ?p)
                                          (shared-property ?c2 ?cfs ?p))
                                       (applicable-property ?c1 ?p)))
                       (:for-some ?p   (:and (Description-Property ?p)
                                       (shared-property ?c1 ?cfs ?p)
                                  (:not (applicable-property ?c2 ?p))))))
 (defrelation most-on-point
 :domain case
 :range case
 :is (:satisfies (?c  ?cfs)
        (:and (case ?c)
              (case ?cfs)
              (neq ?c ?cfs)
              (:for-all ?c1 (:implies (:and (case ?c1)
                                            (neq ?c1 ?cfs))
                                      (:not (more-on-point ?c1 ?c ?cfs)))))))
```

```
;;;; Recuperacion
;;;(retrieve ?c (:and (case ?c) (most-on-point ?c consulta)))
;;;(ask (more-on-point case1 case2 consulta))


(defun shared-property (c1 c2 p)
  (if (member p (intersection
 (obtener_atributos (eval `(fi ,c1))) (obtener_atributos (eval `(fi ,c1)))))
     'T   'NIL  ))
(defrelation shared-property :predicate ((c1 c2 p) (shared-property c1 c2 p)))
;(ask (shared-property 'case4 'consulta 'has-result))
(defun not-shared-property (c1 c2 p)
  (if (member p (intersection (obtener_atributos c1) (obtener_atributos c2)))
      '0  '1 ))
(defun neq (a b) (not (eq a b)))

;; criterio de relevancia para recuperacion por síntomas. Casos con (al menos)
;; los mismos sintomas que otro. Si c1 tiene un sintoma c2 tambien lo tiene.
;;;El usuario podria hacer la siguiente consulta usando el lenguaje de
;; consultas de Loom, y despues definir un criterio de relevancia usando la
:: consulta en la clausula satisfies como se muestra debajo.
;;;En este caso el criterio es de los predefinidos en el sistema
(retrieve (?c1 ?c2)
         (:and
          (case_with_description ?c1)
          (case_with_description ?c2)
          (neq ?c1 ?c2)
          (:for-some (?d1 ?d2)
                    (:and
                     (has-description ?c1 ?d1)
                     (has-description ?c2 ?d2)
                     (:for-all ?s (:implies
                                (has-symptom ?d1 ?s)
                                (has-symptom ?d2 ?s)))))))
(implies same-symptoms relevanceCriteria)

;; Recuperación por Terminos de similitud
(defconcept similarityTerm
    :is-primitive CBROnto-concept
    :annotations ((documentation "the class of all the concepts representing
 similarity between individuals. These concepts are created to do selection by
 classifying them into the subsumption hierarchy. ")))
```

1.6.1 Medidas y funciones de similitud

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;; funciones de similitud ;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defconcept similarityFunction
    :is-primitive CBROnto-concept
    :annotations ((documentation "the class of all the individuals representing
```

```
a similarity function to compute a similarity value. There will be three
subconcepts,local, global and structural according to the kind of function. )))

(defconcept contentssimilarityFunction
    :is-primitive similarityFunction
      :annotations ((documentation "the class of all the individuals
representing a content similarity functions that can be either local or global
similarity functions. ")))

(defconcept localsimilarityFunction :is-primitive contentssimilarityFunction
           :annotations ((documentation "Local Similarity Functions are used to
compute similarity between simple individuals, i.e, slotless individuals,
either from predefined  types as numbers, strings, .. or not..
These functions are typically associated to the contents components of
similarity measures used in simple individuals.  ")))

(defconcept globalsimilarityFunction :is-primitive contentssimilarityFunction
            :annotations ((documentation "Global Similarity Functions are used
to compute similarity between structured individuals, i.e, individuals with
slots. Global similarity functions indicate how to combine the similarity
values computed (recursively) for the fillers of each slot in the individual.
These functions are typically associated to the contents components of
similarity measures used in structured individuals.  ")))

(defconcept positionsimilarityFunction :is-primitive similarityFunction
            :annotations ((documentation "Position SimilarityFunctions are used
to compute similarity between individuals (structured or simple) by the
position they  are in the domain model. These functions are typically
associated to the position component of similarity measures.")))

;; Relaciones
(defrelation funcion-name  ;;para enlazar con el nombre de la funcion.
    :is-primitive (:and CBROnto-relation (:domain similarityFunction))
    :annotations ((documentation "Link a similarityFunction with the name
 of the lisp function that implements it.")))

(defrelation parameters
       :is-primitive (:and CBROnto-relation (:domain similarityFunction)))

(defrelation contents  ;; similitud por contenidos.
       :is-primitive (:and CBROnto-relation (:range similarityFunction))
      :annotations ((documentation "Link a similarityMeasure with the
similarity Funtion used to compute the similarity by the contents of the
attributes of the individual, i.e., how it relates with other individuals.")))

(defrelation position  ;; similitud por posicion
    :is-primitive (:and CBROnto-relation (:range similarityFunction))
    :annotations ((documentation "Link a similarityMeasure with the similarity
Funtion used to compute the similarity by the position of the individual in the
domain model, i.e., the concepts it belongs to.")))

(defrelation combination  ;; combinacion de las dos anteriores.
```

```
       :is-primitive (:and CBROnto-relation (:range similarityFunction))
       :annotations ((documentation "Link a similarityMeasure with the
similarity Funtion used to combine the contents and position similarity
components.")))

;;;;;;;;;;;;;;; INSTANCIAS PARA LAS FUNCIONES LOCALES
(tell (:about isalto1 localsimilarityFunction
             (funcion-name 'salto) (parameters '(1))
             ;; lista con los parametros de la funcion cuyo nombre se indica
             ;; en funcion-name.
             (documentation " The salto similarity funcion is a local function
to compare two numbers. It Returns 1 if the difference between the numbers is
less or equal than a threshold. The threshold is included as a parameter. Can
create other instantiation with a different parameter.")))

;;;;;;; funcion salto predefinida para comparar números ;;;;;;;;;;
;;;;;;  devuelve 1 si la distancia entre los numeros es menor o
;;;;;;  igual que el umbral.
(defun salto (umbral num1 num2) (if (> (abs (- num1 num2)) umbral) '0 '1 ))

(tell (:about iigual localsimilarityFunction
      (funcion-name 'igual)
      (documentation " The igual similarity funcion is a local function to
               compare any two objects: numbers, symbols, strings, individuals,
       etc. Returns 1 if the objects are equal and 0, otherwise.")))

;; Función igual devuelve 1 si los dos individuos son iguales y 0, e.o.c.
(defun igual (i1 i2) (if (equal i1 i2) 1 0) )

(tell (:about iintervalo localsimilarityFunction
      (funcion-name 'intervalo)
      (parameters '(2000000))
      (documentation " The igual similarity funcion is a local function to
            compare numbers. The lenght of the interval is required")))

;; función de similitud local para intervalos numericos.
;; Sim (valor1, valor2) = 1 - (|valor1 - valor2| / longitud del intervalo
numérico)
;;; Recibe como parámetro la longitud del intervalo dentro del cual estamos
;;; comparando a los dos valores.
(defun intervalo (longitud num1 num2)
  (- 1 (/ (abs (- num1 num2)) longitud)))

;;La instancia itable es de guia, para conocer el nombre de la funcion.
;; Luego se creará una instancia para cad tabla que queramos que se consulte
;; particularizándola en el slot parameters.

(tell (:about itable localsimilarityFunction
    (funcion-name 'table-lookup)
   (documentation " The table mode compare any two objects: numbers, symbols,
strings, individuals, etc. Returns the value stored in the table given as a
parameter (required).")))
```

```
(defun table-lookup (t1 i1 i2)
  (let* ((p1 (position i1 (car t1))) ;; num de fila
         (p2 (position i2 (car t1))) ;; num de columna.
         (fila nil) (col 0)  )
(if (and (not (NULL p1))(not (NULL p2)))
    (progn
      (setq fila (elt (cdr t1) p1))    ;; obtengo la fila.
      (setq col (elt fila p2))
      ))
col))

;;;; Ejemplo de instancia para los colores
(tell (:about itableColor localsimilarityFunction (funcion-name 'table-lookup)
      (parameters
              '(((IDARK_GRAY IDARK_BLUE IBLUE IBLACK IDARK_RED)
               (1    0.5    0      0.5    0.2)
               (0.5  1      0.9    0.5    0.2)
               (0    0.9    1      0      0)
               (0.5  0.5    0      1      0.5)
               (0.2  0.2    0      0.5    1)))))

(tell (:about iCompareString localsimilarityFunction
  (funcion-name 'CompareString)
  (documentation "Function used to compare strings in a non sensitive way. ")))

;;; La funcion CompareString no diferencia entre mayusculas y minusculas:
;;;;;; (CompareString "Pepe" "pepe")  1
(defun CompareString (s1 s2)
  (if (string= (string-downcase s1) (string-downcase s2))  1 0))

(tell (:about iMaxCad localsimilarityFunction
    (funcion-name 'MaxCad)
(documentation "Function used to compare strings based on the substrings they
share. ")))

;; No distinguimos entre may y min. Si queremso distinguir cambiar string-equal
por string=
;; Ejemplos
;; (MaxCad "pepe" "Pepa") ¾ (MaxCad "pepe" "Pepito") ½ (MaxCad "pepe" "PPEP") ¾

(defun MaxCad (s1 s2)
  (if (equal s1 s2) 1  (/ (MaxSubcadena s1 s2) (max (length s1) (length s2)))))

(defun MaxSubcadena (s1 s2)
  (let ((l1 (- (length s1) 1))(l2 (- (length s2) 1))(long 0))
  (loop for i from 0 to l1 do
  (loop for j from (+ l1 1) downto i do
  (loop for k from 0 to l2 do
      (loop for l from (+ l2 1) downto k
         do   (if (string-equal s1 s2 :start1 i :end1 j :start2 k :end2 l)
                 (setq long (max long (- j i))))))))   long))
```

```
;;;;;;;; INSTANCIAS PARA LAS FUNCIONES de POSICION ;;;;;;;;;;;;;;;;;;;;;;;
(tell (:about iconstant1 positionsimilarityFunction
             (funcion-name 'constant)
             (parameters '(1))
             (documentation "función de similitud que, independientemente de
los individuos comparados, devuelve un valor constante 1. Es util para anotar
los conceptos con valores de similitud constantes y crecientes con la
profundidad [Bergmann]")))

(defun constant (r) r)
 ;; "función de similitud que, independientemente de los individuos comparados,
devuelve un valor constante que coincide con el parametro. Es util para anotar
los conceptos con valores de similitud constantes y crecientes con la
profundidad [Bergmann]")

(tell (:about ideep_basic positionsimilarityFunction
             (funcion-name 'fdeep_basic)
             (documentation "computa como la relación existente entre la
profundidad del concepto más específico que contiene a los dos individuos
comparados ( (i1,i2)) y la profundidad máxima de la jerarquía.")))

(defun fdeep_basic (i1 i2)
  (/ (profundidad_concepto  (compute-conjunction-concept (intersection
       (remove (find-concept 'okbc-individual) (get-types i1))
       (remove (find-concept 'okbc-individual) (get-types i2)))))
   (profundidad_kb)))

(tell (:about ideep positionsimilarityFunction
             (funcion-name 'fdeep)
             (documentation "De forma similar a la anterior, la función de
similitud profundidad computa la similitud según la relación que exista entre
la profundidad del concepto más específico que contiene a los dos individuos
comparados y la del individuo más profundo de los dos. ")))

(tell (:about icosine positionsimilarityFunction
             (funcion-name 'simescalar)
             (documentation "La función de similitud coseno se basa en la
función de similitud entre conceptos definida en [González-Calero97] e
inspirada en el modelo del espacio vectorial")))

;;De forma similar a la anterior, la función de similitud profundidad computa
la similitud según la relación que exista entre la profundidad del concepto más
específico que contiene a los dos individuos comparados y la del individuo más
profundo de los dos.

;; producto escalar de los vectores que representan los conceptos comunes.
;;(sim-escalar (fi idark) (fi idark_red)) 0.9258201

(defun simescalar (i1 i2)
     (if (and (instance-p  i1) (instance-p i2))
      (let* ((super1 (get-types i1))
```

```
            (super2 (get-types i2))
            (comunes (remove-if-not #'(lambda (x) (member x super2)) super1))
            (sim 0) ;;inicializo sim a 0
            )
       (setf sim
(min (/ (length comunes) (* (sqrt (length super1)) (sqrt (length super2))))
1)))
    ;; else. Si no son instancias devuelve 0
    0))
;;;;;;;;;; INSTANCIAS PARA LAS FUNCIONES GLOBALES ;;;;;;;;;;;;;;;;;;;;;;;;;;;
(tell (:about imedia globalsimilarityFunction
             (funcion-name 'media)
             (documentation  " The media similarity function combines a list of
values by the arithmetic average between them.") ) )
;; recibe una lista de valores y computa la media aritmética.
(defun media (lista)  (/ (reduce #'+ lista) (length lista)) )

(tell (:about imedia-ponderada globalsimilarityFunction
     (funcion-name 'media-ponderada)  (parameters nil)
     (documentation  " The weighted average similarity function combines a list
of values by the arithmetic average between them weighted correspondingly.It
requires the weights vector to be specified as a parameter.")))

(tell (:about imedia-ponderada-con globalsimilarityFunction
             (funcion-name 'media-ponderada)
             (parameters '((0.8 0.2)))
 ;; el primer parametro es para contenidos y el segundo para posicion
 ;; quiero tener mas en cuenta el contenido que su posicion.
  (documentation  " The weighted average similarity function combines a list of
values by the arithmetic average between them weighted correspondingly.It
requires the weights vector to be specified as a parameter.")))

(tell (:about imedia-ponderada-pos globalsimilarityFunction
      (funcion-name 'media-ponderada)
      (parameters '((0.2 0.8)))
      ;; el primer parametro es para contenidos y el segundo para posicion
      ;; quiero tener mas en cuenta la posicion que el contenido.
     (documentation  " The weighted average similarity function combines a list
of values by the arithmetic average between them weighted correspondingly.It
requires the weights vector to be specified as a parameter.") ))

;; recibe una lista de pesos y una lista de valores y computa la media
ponderada de ellos.
(defun media-ponderada (listapesos listavalores)
  (if (not (eq (length listapesos) (length listavalores)))
      (media listavalores) ;; si no hay los mismos pesos que valores hago la
media normal.
    ;; else
    (let ((suma (loop for x in listapesos
                sum (* x (pop listavalores)))))
      (/ suma (reduce #'+ listapesos)) ;; divido por la suma de los pesos.
      )))
```

```
(tell (:about iminkowski_with1 globalsimilarityFunction
             (funcion-name 'minkowski)
             (parameters '(1))
             (documentation  " The iminkowski_with1 similarity function
combines a list of values by the minkowski metric using r=1")))

;;;Minkowski.
(defun minkowski (r lista)
  ;(expt x (/ 1 r))  ;; raiz r-esima de x.
  (/ (expt (reduce #'+ (mapcar #'(lambda (x) (expt x r)) lista)) (/ 1 r))
(length lista)))

(tell (:about ieuclidea globalsimilarityFunction
      (funcion-name 'euclidea)
      (documentation  " The euclidea similarity function combines a list of
values by the minkowski metric using r=2")))
;;;Euclidea (Minkowski con r=2).
(defun euclidea (lista) (minkowski 2 lista))

(tell (:about iminkowski_with3 globalsimilarityFunction
             (funcion-name 'minkowski)
             (parameters '(3))
             (documentation  " The iminkowski_with3 similarity function
combines a list of values by the minkowski metric using r=3")))

(tell (:about iminkowski globalsimilarityFunction
      (funcion-name 'minkowski)
      (documentation  " The iminkowski similarity function combines a list of
values by the minkowski metric. It requires the r parameter. ")))

(tell (:about iminkowski-ponderada globalsimilarityFunction
             (funcion-name 'minkowski-ponderada)
             (parameters nil)
             (documentation  " The iminkowski_-ponderada similarity function
combines a list of values by the minkowski metric. It requires the r parameter
as well as the weights vector.")))

;;;Minkowski ponderada.
;;; ejemplo
;; (minkowski-ponderada 1 '(1 2 2) '(1 1 1))

(defun minkowski-ponderada (r listapesos lista)
;;(expt x (/ 1 r))  ;; raiz r-esima de x.
   (if (not (eq (length listapesos) (length lista)))
      (minkowski lista) ;; si no hay los mismos pesos que valores hago la
normal.
     ;; else
    (/
     (expt (reduce #'+ (mapcar #'* listapesos (mapcar #'(lambda (x) (expt x r))
lista)))
          (/ 1 r))
```

```
     (reduce #'+ listapesos))
 ))

(tell (:about imaximum globalsimilarityFunction
      (funcion-name 'maximo)
      (documentation  "The maximum similarity function combines a list of
values by extracting the maximum of them.")))

(tell (:about iminimum globalsimilarityFunction
      (funcion-name 'minimo)
      (documentation  "The minimum similarity function combines a list of
values by extracting the minimum of them.") ))

(tell (:about iclarck globalsimilarityFunction
      (funcion-name 'clarck)
      (documentation  " Clarck function.")))

;;; maximo
(defun maximo (lista) (reduce #'max lista) )

;;; minimo
(defun minimo (lista)(reduce #'min lista) )

;;;Máximo con pesos.
(defun maximo-ponderada (listapesos lista)
  (if (not (eq (length listapesos) (length lista)))
      (maximo lista)
    ;;else
    (maximo (mapcar #'* listapesos lista))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;; medidas de similitud ;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defconcept similarityMeasure
    :is-primitive  (:and CBROnto-concept
                  (:all contents  contentssimilarityFunction)
                  (:at-most 1 contents)
                  (:all position        positionsimilarityFunction)
                  (:at-most 1 position)
                  (:all combination     globalsimilarityFunction)
                  (:at-most 1 combination)
                  (:the similarity-type similarityTypes)   )
    :annotations
    ((documentation "the class of all the individuals representing
         a similarity measure to compare instances of a concept.")))

(defrelation currentMeasure :range similarityMeasure
    :annotations ((documentation "this relation links a concept
with the similarity measure that will be used to compute the similarity
value among two concept instances.")))
```

```
;; la relacion currentMeasure se usarß como anotacion en un concepto, porque
las medidas de similitud
;; se asocian a los conceptos y no a las instancias.

(defrelation userMeasure
    :range similarityMeasure
    :annotations ((documentation "this relation links a concept
with the user defined  similarity measure that will be used to compute the
similarity
value among two concept instances during the current user interaction. ")))
;; la relacion userMeasure tambien se usara como anotacion en un concepto

(defrelation oldMeasure
    :range similarityMeasure
    :annotations ((documentation "this relation links a concept
with an old  similarity measure that won't be used while a currentMeasure
exists. ")))
;; Medida por defecto. Puede valer si iigual se considera local y global.
(tell (:about default_measure similarityMeasure
             (contents iigual)
             (position ideep)
             (combination imedia-ponderada-con)))

;; En la funcion de contenidos para casos estructurados tendre que usar una
funcion global y no una local.
(tell (:about caseMeasure similarityMeasure
             (contents iminkowski_with1)
             ;;(contents imedia)
             (position ideep)
             (combination imedia)
             ;;(combination iminkowski_with1)
```

1.6.2 Tipos de fallo y estrategias de adaptación y reparación

```
(defconcept Adaptation_Strategy :is-primitive
(:and  CBROnto-Concept
   (:at-least 1 transformation-spec) (:all transformation-spec Transformation)
   (:all strategy-operational-spec Strategy-Operational-Specification)))

(defconcept Repair_Strategy :is-primitive
(:and  CBROnto-Concept
   (:at-least 1 transformation-spec) (:all transformation-spec Transformation)
   (:all strategy-operational-spec Strategy-Operational-Specification)))

(defconcept Transformation :is-primitive  (:and  CBROnto-Concept
              (:the operator Transformation_Operator)
              (:at-most 1 applicability) (:all applicability Applicability)
              (:the weith Number)))
(defconcept Add_Transformation :is
              (:and Transformation (:the operator Add_Operator)
                     (:the add-strategy Add-search-strategy)))
(defconcept Delete_Transformation :is
```

```
              (:and Transformation (:the operator Delete_Operator)))
(defconcept Substitute_Transformation :is
              (:and Transformation (:the operator Substitute_Operator)
                     (:at-most 1 tosubstitute) (:at-most 1 search-strategy)
                     (:all search-strategy Search-Strategy)))

(defconcept Add_Operator :is-primitive Transformation_Operator)
(defconcept  Delete_Operator :is-primitive Transformation_Operator)
(defconcept Substitute_Operator :is-primitive Transformation_Operator)
(tell (:about add Add_Operator))
(tell (:about delete Delete_Operator))
(tell (:about substitute Substitute_Operator))

(defconcept Strategy-Operational-Specification :is-primitive
       (:and CBROnto-Concept
              (:at-most 1 has-method) (:all has-method CBR_Method)
              (:at-most 1 lisp-function) (:all lisp-function Method_Function)
       ))

(defconcept Add-search-strategy :is-primitive (:and CBROnto-Concept
        (:the begin-with Starting-point)
        (:all step Search-Step) (:at-most 1 weight)))
(defconcept Starting-point :is-primitive (:and CBROnto-Concept (:one-of
current-case current-query))))

(defconcept Simple-Search-Strategy :is-primitive Add-search-strategy)
(defconcept Method-Search-Strategy :is-primitive (:and Add-search-strategy
   (:at-most 1 weight) (:the search-method Search-Substitutes-Method)))
(defconcept Applicability :is-primitive CBROnto-Concept)
(defconcept Search-Step :is-primitive (:and CBROnto-Concept
       (:the step-number Number)
       (:the primitive-step Primitive-Step)))
(defconcept Primitive-Step :is-primitive (:and CBROnto-Concept))
(tell (:about role-filler Primitive-Step)) ;; la instancia role-filler hay que
asignarle el atributo role que indica el nombre del atributo cuyo relleno
queremos extraer.
(tell (:about superrelation Primitive-Step)) ;; la instancia superrelation
tiene un atributo rel que indica el nombre de la relación que queremos
generalizar.
(tell (:about siblings-at-level Primitive-Step));; la instancia siblings-at-
level tiene un atributo level que indica el nivel (positivo o negativo) al que
recuperar instancias cercanas.
(defrelation strategy-operational-spec :is-primitive CBROnto-relation)
(defrelation has-method :is-primitive strategy-operational-spec
                      :range CBROnto-Method)
(defrelation lisp-function :is-primitive strategy-operational-spec
                      :range Method_Function)
(defrelation transformation-spec :is-primitive CBROnto-relation :range
Transformation)
(defrelation step :is-primitive CBROnto-relation  :range Search-Step)
(defrelation add-strategy :is-primitive CBROnto-relation  :range Add-search-
strategy)
```

```
(defrelation search-strategy :is-primitive CBROnto-relation :range Search-
Strategy)
(defrelation applicability :is-primitive CBROnto-relation :range
Applicability)
(defrelation tosubstitute :is-primitive CBROnto-relation)
(defrelation step-number :is-primitive CBROnto-relation)
(defrelation primitive-step :is-primitive CBROnto-relation)

;; Tipos de problema. El diseñador definirá los tipos de problema específicos
de
;; una aplicación y para cada uno se anota la estrategia de adaptación que
;; corresponda.
(defconcept Problem_Type :is-primitive CBROnto-Concept
   (:all has_adaptation_strategy Adaptation_Strategy))
(defrelation has_adaptation_strategy
:is-primitive CBROnto-Relation :range Adaptation_Strategy)

;; Igual con los tipos de fallo de reparación y estrategias de reparación
(defconcept Fail_Type :is-primitive CBROnto-Concept
   (:all has_repair_strategy Repair_Strategy))
```

2. Ontología de tareas y métodos

2.1 TAREAS

```
(defconcept CBROnto_Task :is-primitive
       (:and CBROnto-concept
          (:exactly 1 task_name)
          (:at-most 1 task_method)))
;; el metodo elegido para resolver una tarea puede estar seleccionado o no
;; (usando task_method). Otros metodos alternativos se pueden obtener usando la
;; inversa de la competencia de los metodos. (competence)

(defconcept Case_Acquisition_Task
       :is-primitive (:and CBROnto_Task
                           (:filled-by task_method user_resolution_method))
       :annotations ((documentation "Case Acquisition task")))

(defconcept CB_Organization_Task
       :is-primitive (:and CBROnto_Task
                           (:all task_method CB_Organization_Method))
       :annotations ((documentation "Case Base Organization task")))

(defconcept Domain_Model_Adq_Task
       :is-primitive (:and CBROnto_Task
                           (:filled-by task_method user_resolution_method))
       :annotations ((documentation "Domain Model Acquisition task")))

(defconcept Mapping_Task
       :is-primitive (:and CBROnto_Task
                           (:filled-by task_method user_resolution_method))
       :annotations ((documentation "Mapping task")))
```

```
(defconcept CB_Maintenance_Task
       :is-primitive (:and CBROnto_Task
                           (:filled-by task_method user_resolution_method))
       :annotations ((documentation "Case Base Maintenance task")))

;; Tarea de resolución de problemas.
(defconcept CBR_TASK
       :is-primitive (:and CBROnto_Task (:all task_method CBR_METHOD))
       :annotations ((documentation "the class of all the CBR Tasks")))

;; SUBTAREAS DE CBR_TASK

(defconcept Retrieve_Task :is-primitive
  (:and CBR_TASK
       (:filled-by task_name "Retrieve")
       (:all task_method Retrieval_Method)))

(defconcept Reuse_Task :is-primitive
  (:and CBR_TASK
       (:filled-by task_name "Reuse")
       (:all task_method Reuse_Method)))

(defconcept Revise_Task :is-primitive
  (:and CBR_TASK
       (:filled-by task_name "Revise")
       (:all task_method Revise_Method)))

(defconcept Retain_Task :is-primitive
  (:and CBR_TASK
       (:filled-by task_name "Retain")
       (:all task_method Retain_Method)))

(defconcept ObtainCases_Task :is-primitive
  (:and Retrieve_Task
       (:filled-by task_name "Obtain set of cases")
       (:all task_method Obtain_Cases_Method)))

(defconcept AssessSim_Task :is-primitive
  (:and Retrieve_Task
       (:filled-by task_name "Similarity Assessment")
       (:all task_method AssessSim_Method)))

(defconcept Select_Task :is-primitive
  (:and Retrieve_Task
       (:filled-by task_name "Selection")
       (:all task_method Selection_Method)))

(defconcept Copy_Solution_Task :is-primitive
  (:and Reuse_Task
       (:filled-by task_name "Copy Solution")
       (:all task_method Copy_Solution_Method)))
```

```
(defconcept Adapt_Solution_Task :is-primitive
  (:and Reuse_Task
        (:filled-by task_name "Adapt Solution")
        (:all task_method Adapt_Solution_Method)))

(defconcept Select_Strategy_Task :is-primitive
  (:and Adapt_Solution_Task
        (:filled-by task_name "Select Strategy")
        (:all task_method Select_Strategy_Method)))

(defconcept Select_Discrepancy_Task :is-primitive
  (:and Adapt_Solution_Task
        (:filled-by task_name "Select Discrepancy")
        (:all task_method Select_Discrepancy_Method)))

(defconcept Modify_Solution_Task :is-primitive
  (:and Adapt_Solution_Task
        (:filled-by task_name " Modify_Solution ")
        (:all task_method Modify_Solution_Method)))

(defconcept Apply_Transformation_Task :is-primitive
  (:and Modify_Solution_Task
        (:filled-by task_name " Apply_Transformation ")
        (:all task_method Apply_Transformation_Method)))

(defconcept Local_Revision_Task :is-primitive
  (:and Modify_Solution_Task
        (:filled-by task_name " Local_Revision ")
        (:all task_method Local_Revision_Method)))

(defconcept Evaluate_Task :is-primitive
  (:and Revise_Task
        (:filled-by task_name "Evaluate task")
        (:all task_method Evaluate_Method)))

(defconcept Repair_Task :is-primitive
  (:and Revise_Task
        (:filled-by task_name "Repair task")
        (:all task_method Repair_Method)))

(defconcept Select_Repair_Strategy_Task :is-primitive
  (:and Repair_Task
        (:filled-by task_name "Select repair strategy")
        (:all task_method Select_Repair_Strategy_Method)))

(defconcept Apply_Repair_Strategy_Task :is-primitive
  (:and Repair_Task
        (:filled-by task_name "Apply repair strategy")
        (:all task_method Apply_Repair_Strategy_Method)))

(defconcept Retain_Case_Task :is-primitive
  (:and Retain_Task
```

```
       (:filled-by task_name "Retain Case")
       (:all task_method Retain_Case_Method)))

(defconcept Retain_Knowledge_Task :is-primitive
  (:and Retain_Task
       (:filled-by task_name "Retain Knowledge")
       (:all task_method Retain_Knowledge_Method)))

(defconcept Retain_retrieval_knowledge_Task :is-primitive
  (:and Retain_Knowledge_Task
       (:filled-by task_name "Retain retrieval Knowledge")
       (:all task_method Retain_retrieval_knowledge_Method)))

(defconcept Retain_reuse_knowledge_Task :is-primitive
  (:and Retain_Knowledge_Task
       (:filled-by task_name "Retain reuse Knowledge")
       (:all task_method Retain_reuse_knowledge_Method)))

(defconcept Retain_revise_knowledge_Task :is-primitive
  (:and Retain_Knowledge_Task
       (:filled-by task_name "Retain revise Knowledge")
       (:all task_method Retain_revise_knowledge_Method)))
;;; Relaciones
(defrelation task_name
       :is-primitive  (:and CBROnto-relation (:range string))
       :annotations ((documentation "the task-name relation links each
 task instance with its textual name to be used for the system designer.")))

(defrelation task_method
      :is-primitive CBROnto-relation
      :annotations ((documentation "the task_method relation links each
 task instance with the method used to solve the task.")))
```

2.2 MÉTODOS

```
(defconcept CBR_METHOD :is-primitive
  (:and  CBROnto-concept
         (:exactly 1 method_name)
         (:exactly 1 method_informal_description)
         (:all competence CBR_Task)
         (:all method_application_requirements Application_Requirements)
         (:all method_input Method_Input)
         (:all method_output Method_Output)
         (:exactly 1 operational_specification)
         (:all operational_specification Method_Function))
  :exhaustive-partitions $METHODTYPES$
  :annotations ((documentation "the class of all the CBR Methods"))  )

(defconcept DECOMPOSITION_METHOD
  :is-primitive (:and CBR_METHOD (:at-least 1 subtask) (:all subtask CBR_TASK))
  :in-partition $METHODTYPES$
```

```
  :annotations ((documentation "the class of all the Decomposition Methods,
i.e. those that decompose a task in subtasks to be solved by other methods.")))

(defconcept CYCLE_METHOD :is-primitive DECOMPOSITION_METHOD) ;; Ciclamos un
numero de veces.

(defconcept RESOLUTION_METHOD
    :is-primitive (:and CBR_METHOD
                        (:at-most 0 subtask))
    :in-partition $METHODTYPES$
    :annotations ((documentation "the class of all the Resolution Methods, i.e.
those that directly solve a certain task.")))

(defconcept Retrieval_Method
    :is-primitive  (:and CBR_METHOD(:all competence Retrieve_Task))
    :annotations ((documentation "the class of all the individuals representing
a method to retrieve cases from the case base. ")))

(defconcept Reuse_Method
    :is-primitive  (:and CBR_METHOD (:all competence Reuse_Task))
    :annotations ((documentation "the class of all the individuals representing
a method to reuse or adapt cases. ")))

(defconcept Revise_Method
    :is-primitive  (:and CBR_METHOD (:all competence Revise_Task))
    :annotations ((documentation "the class of all the individuals representing
a method to revise adapted cases ")))

(defconcept Retain_Method
    :is-primitive  (:and CBR_METHOD (:all competence Retain_Task))
    :annotations ((documentation "the class of all the individuals representing
a method to retain new cases.")))

;; Subconceptos
(defconcept ObtainCases_Method :is-primitive (:and Retrieval_Method))
(defconcept AssessSim_Method :is-primitive  (:and Retrieval_Method))
(defconcept Select_Method :is-primitive  (:and Retrieval_Method))
(defconcept Copy_Solution_Method :is-primitive (:and Reuse_Method))
(defconcept Adapt_Solution_Method :is-primitive (:and Reuse_Method))
(defconcept Select_Strategy_Method :is-primitive (:and Adapt_Solution_Method))
(defconcept Select_Discrepancy_Method :is-primitive (:and
Adapt_Solution_Method))
(defconcept Modify_Solution_Method :is-primitive (:and Adapt_Solution_Method))
(defconcept Apply_Transformation_Method :is-primitive (:and
Modify_Solution_Method))
(defconcept Local_Revision_Method :is-primitive (:and Modify_Solution_Method))
(defconcept Evaluate_Method :is-primitive (:and Revise_Method))
(defconcept Repair_Method :is-primitive (:and Revise_Method))
(defconcept Select_Repair_Strategy_Method :is-primitive (:and Repair_Method))
(defconcept Apply_Repair_Strategy_Method :is-primitive (:and Repair_Method))
(defconcept Retain_Case_Method :is-primitive (:and Retain_Method))
(defconcept Retain_Knowledge_Method :is-primitive (:and Retain_Method))
```

```
(defconcept Retain_retrieval_knowledge_Method :is-primitive (:and
Retain_Knowledge_Method))
(defconcept Retain_reuse_knowledge_Method :is-primitive
  (:and Retain_Knowledge_Method))
(defconcept Retain_revise_knowledge_Method :is-primitive
  (:and Retain_Knowledge_Method))
(defconcept CB_Organization_Method :is-primitive CBR_Method)

(defconcept Method_Input
    :is-primitive (:and CBROnto-concept
                        (:the knowledge_requirements Knowledge_Requirements)
                        (:the sequence_requirements Sequence_Requirements)
                        (:the design_requirements Design_Requirements)
                        (:the parameter_requirements Parameter_Requirements))
    :annotations ((documentation "the class of all the input descriptions. Its
instances are used to describe the inputs of a method and the conditions to be
satisfied.")))

(defconcept Method_Output   :is-primitive (:and CBROnto-concept)
    :annotations ((documentation "the class of all the output descriptions. Its
instances are used to describe the outputs of a method.")))
(defconcept Application_Requirements  :is-primitive (:and CBROnto-concept)
    :annotations ((documentation "the class of all the application requirements
that are used to check the applicability conditions in the methods..")))

;;;;; relaciones ;;;
(defrelation method_name :is-primitive (:and CBROnto-relation (:range string))
  :annotations ((documentation "the method-name relation links each
 method instance with its textual name to be used for the system designer.")))

(defrelation method_informal_description :is-primitive
  (:and CBROnto-relation (:range string))
  :annotations ((documentation "the method_informal_description relation links
each  method instance with its textual description to be used for the system
designer. The description should include the method description at the
knowledge level.")))

(defrelation method_input :is-primitive
  (:and CBROnto-relation (:range Method_Input))
  :annotations ((documentation "the  method_inputs relation links each
 method instance with a Method_Input instance, that specify the method inputs
 and the conditions to be satisfied by them.")))

(defrelation method_output :is-primitive
       (:and CBROnto-relation (:range Method_Output))
  :annotations ((documentation "the  method_output relation links each method
instance with a Method_Output instance, that specify the method outputs.")))

(defrelation knowledge_requirements :is-primitive
  (:and CBROnto-relation (:range Knowledge_Requirements))
  :annotations ((documentation "the knowledge_requirements relation links each
```

```
 method instance with a Knowledge_Requirements instance, that specify the
method knowledge requirements.")))

(defrelation sequence_requirements :is-primitive
  (:and CBROnto-relation (:range Sequence_Requirements)))

(defrelation design_requirements :is-primitive
  (:and CBROnto-relation (:range Design_Requirements)))

(defrelation parameter_requirements :is-primitive
  (:and CBROnto-relation (:range Parameter_Requirements))

(defrelation competence
  :is-primitive (:and CBROnto-relation (:range CBROnto_Task))
  :annotations ((documentation "the  competence relation links each
 method instance with a Competence instance, that specify the method
constraints
 across the inputs and outputs that state what the method can accomplish. These
constraints are guaranteed to be true if the method executes successfully. It
communicates some notion of method semantics.")))

(defrelation operational_specification
  :is-primitive (:and CBROnto-relation  (:range Method_Function))
  :annotations ((documentation "the  operational_specification relation links
each  method instance with a Method_Function instance, that specify the method
functional specification, i.e., how the method achieves its task.")))

(defrelation subtask
  :is-primitive (:and CBROnto-relation (:range CBROnto-Task))
  :annotations ((documentation "The subtask relation links each method with its
subtasks. It is only used for documentation purposes so that the control flow
is not explicitly represented in the method.")))

(defconcept Method_Function
  :is-primitive (:and CBROnto-Concept (:exactly 1 function_name))
  :annotations ((documentation " The class of all the instances that
declaratively represents Lisp functions accomplishing some methods. They are
used to represent the  method operational_specification as in our scheme the
control flow is not explicitly represented.")))

(defrelation function_name
 :is-primitive (:and  CBROnto-Relation (:range string))
 :annotations ((documentation "the operational_specification relation links
each method instance with the name of the lisp function that achieves the
task")))

(defrelation type_of_application
 :annotations ((documentation "type of application links each method with
the type of application for which it is specially well suited. That can be
annotated or learned.")))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;; INDIVIDUOS CANÓNICOS DE LA ONTOLOGÍA DE TAREAS Y MÉTODOS ;;;;;;
(tell  (:about iCBR_task CBR_TASK (task_name "CBR Task")
       (task_method iCBR_Method)
       (documentation "Tarea principal de resolución de problemas. Tiene
asignado el método CBR para su resolución")))
;; El individuo iCBR_Method representa el m%todo de resoluci□n de problemas
llamado razonamiento basado en casos. Es un m%todo de descomposici□n ya que
para resolver un problema realiza 4 subtareas: retrieve, reuse, revise y
retain.

(tell (:about iCBR_Method CBR_Method Decomposition_Method
      (method_name "CBR Method")
      (method_informal_description "This method applies CBR to resolve a
problem. It decompose the resolution task into four subtasks, namely retrieve,
reuse, revise y retain")
       (competence iCBR_task)
       (method_application_requirements iCBR_Application)
       (method_input iCBR_Inputs) (method_output iCBR_Output)
       (subtask iRetrieve_task) (subtask iReuse_task)
       (subtask iRevise_task) (subtask iRetain_task)
       (operational_specification iCBR_fun_spec)))

(tell (:about  iCBR_fun_spec Method_Function (function_name 'CBRCycle) ))
;; -----------  SUBTAREAS ------------
(tell (:about iRetrieve_task Retrieve_Task))
(tell (:about iReuse_task Reuse_Task))
(tell (:about iRevise_task Revise_Task))
(tell (:about iRetain_task Retain_Task))

;; ------------ MÐTODOS ASOCIADOS A LAS SUBTAREAS --------------------
;; En el apéndice C se incluye una descripción textual de toda la biblioteca de
métodos. Por razones de espacio en este apéndice sólo incluimos algunos métodos
para dar idea de la representación en Loom. El resto de los métodos se han
representado de forma similar.

(tell (:about user_resolution_method CBROnto_Method)
;; RECUPERACIÔN
;; Recuperación aleatoria
(tell (:about iRandom_Retrieval_Method Random_Retrieval_Method
       Resolution_Method(competence IRETRIEVE_TASK)
       (method_name "random") (method_input iRandom_Inputs)
       (method_informal_description "This method retrieve a case randomly")
       (operational_specification iRandom_Retrieval_Function) ))

(tell (:about iRandom_Retrieval_Function Method_Function
             (function_name 'Random_Retrieval_Function)))
;; Recuperacion manual por el usuario.
(tell (:about iUser_Retrieval_Method
             User_Retrieval_Method  Resolution_Method
             (competence IRETRIEVE_TASK) (method_name "user retrieval")
             (method_input iUserRetrieve_Inputs)
```

```
            (method_informal_description "This method retrieve a case manually
by the user")
            (operational_specification iUser_Retrieval_Function)
            ))

(tell (:about iUser_Retrieval_Function Method_Function
            (function_name 'User_Retrieval_Function)))

;; Recuperacion por cómputo de similitud.
(tell (:about iRetrieve_computational_method Decomposition_Method
    (method_competence iretrieve_task) ;; se reconoce como Retrieval_Method
    (method_name "computational")
    (method_application_requirements iComputationApplication_Requirements)
    (method_input iComputation_Inputs) (method_output iRetrieval_Output)
    (method_informal_description "This method retrieve cases based on their
similarity values within the query.")
   (functional_specification iComputationalFunction)
    (subtask     iobtain_cases)(subtask     icompute_similarity)     (subtask
iselect_best)))

;; Recuperacion por clasificación.
(tell (:about iRetrieve_instance_classification_method  Decomposition_Method
     (method_name "instance classification")
     (method_informal_description "This method retrieve instances according to
   the distance in the representational structure ")
        (method_competence iretrieve_task)
        (method_application_requirements iClassification_Application)
        (method_input iClassification_Inputs)
         (functional_specification     iInstanceFunction)           (subtask
irecognize_task) (subtask iselect_case))

(tell (:about iRetrieve_concept_classification_method Decomposition_Method
        (method_name "concept classification") (method_competence iretrieve_task)
        (method_application_requirements iClassification_Application)
        (method_input iClassification_Inputs) (method_output
   iRetrieval_Output)
        (functional_specification iConceptFunction)
             (subtask iclassify_task)(subtask iselect_case))

;; Recuperacion por criterios de relevancia.
(tell (:about iRetrieve_RelCriteria_Method
            Decomposition_Method
            (method_competence iretrieve_task)
             ;; se reconocerá como instancia de Retrieval_Method
            (method_name "Relevance Criteria")
             (method_application_requirements iRelCriteria_Application)
            (method_input iRelCriteria_Inputs)
            (method_informal_description "This method retrieve cases based on
a given similarity criteria formulated as a logic predicate.")
            (functional_specification iRelCriteriaFunction)
            (subtask iretrieve_relevance_task) (subtask iselect_case)))
```

```
;; Recuperacion por términos de similitud.
   (tell (:about iRetrieve_SimilarityTerms_Method Decomposition_Method
              (method_competence iretrieve_task)
              (method_name "Relevance Criteria")
               (method_application_requirements iRelCriteria_Application)
              (method_input iRelCriteria_Inputs)
              (method_informal_description "This method retrieve cases based on
a given similarity criteria formulated as a logic predicate.")
              (functional_specification iSimTermsFunction)
               (subtask iFilter_Cases) (subtask iCompute_simTerms_Task)
   (subtask iselect_case)))
;; Adaptación
(tell (:about iAdapt_Specialized_Method
             Decomposition_Method (method_competence ireuse_task)
             (method_name "Specialized Adaptation Method")
              (method_application_requirements iAdapt_Specialized_Application)
             (method_input iAdapt_Specialized_Input)
              (method_output iReuse_Output)
             (functional_specification iAdapt_Specialized_Function)
             (subtask iretrieve_adaptation_strategy_task)
             (subtask ifind_adaptation_actions_task)
             (subtask imodify_solution_task)))

   (tell (:about iAdapt_by_Substitution_Method Decomposition_Method
              (method_competence ireuse_task)
               (method_name "Specialized Adaptation Method")
               (method_application_requirements iReuse_Application)
              (method_input iAdapt_by_Substitution_Inputs)
              (functional_specification iAdapt_by_Substitution_Function)
              (subtask isubstitution_strategy_task)
               (subtask ifind_adaptation_actions_task)
              (subtask imodify_solution_task)))

;; Aprendizaje
(tell (:about iRetain_Method
             Decomposition_Method
             (method_competence iretain_task)
              ;; se reconocerá como instancia de Retrieval_Method
             (method_name "Retain Method")
              (method_application_requirements iRetain_Application)
             (method_input iRetain_Inputs) (method_output iRetain_Output)
             (functional_specification iRetainFunction)
             (subtask iretrieval_knowledge_retain_task)
             (subtask iretain_adaptation_knowledge_task)))
```

APÉNDICE C: LISTADO DE LA BIBLIOTECA DE MÉTODOS DE CBRONTO

Para cada método se incluye una descripción breve, su competencia, las subtareas que genera si es un método de descomposición, su salida, sus requisitos de aplicabilidad y sus entradas: requisitos paramétricos, de diseño, de conocimiento y de secuencia.

---

iCBR_Method: razonamiento basado en casos
Competencia: iCBR_Task (CBR_Task)
Subtareas: iRetrieve_task (Retrieve_Task), iReuse_task (Reuse_Task), iRevise_task (Revise_Task), iRetain_task (Retain_Task)
Salida: (iRetrieval_Output, iReuse_Output, iRevise_Output, iRetain_Output). Hace referencia a la salida que se produce al resolver cada una de las tareas. Sólo la primera es obligatoria.
El método puede generar más resultados de salida en forma de conocimiento que se añade al modelo de dominio (en el aprendizaje). Pero no se refleja como salida explícita.
Aplicabilidad: (iCBR_Application) nombre de la base de casos representada por un concepto tipo de caso). La base de casos debe ser no vacía, de casos con descripción.
Entrada: (iCBR_Inputs)
Paramétricos: (iCBR_Parameter) consulta (query), nombre de la base de basos (casebase)
Conocimiento: (iCBR_Knowledge) base de casos (subconcepto de Case)
Diseño: (iCBR_Design) --
Secuencia: --

---

ido_Nothing_Method: método que no hace nada, se usa para asociarlo a las tareas que no se van a resolver. Su competencia son las tareas opcionales de las secuencias de tareas.
Competencia: iReuse_Task, iRevise_Task, iRetain_Task, iAutomatic_Revision_task, iUser_Revision_task, ilocal_revisión_task, iretain_retrieval_knowlege_task, iretain_reuse_knowledge_task e iretain_revise_knowledge_task ….
Subtareas: --
Salida: devuelve como salida el individuo que recibe como salida del método anterior de una secuencia.
Aplicabilidad: --
Entrada: --

RECUPERACIÓN

---

iRetrieve_Computational_Method: recuperación de casos por cómputo de similitud numérica.
Competencia: iretrieve_task
Subtareas: iobtain_cases (ObtainCases_Task), icompute_similarity (AssessSim_Task), iselect_best (Select_Task)
Salida: (iRetrieval_Output) atributo retrieved_cases (lista ordenada de uno o más casos recuperados)
Aplicabilidad: (iComputation_Application)
case_base- heredado. Base de casos no vacía, de casos con descripción. Se añade la propiedad de que la base de casos debe tener un tamaño no muy grande.
Entrada (iComputation_Inputs)
Paramétricos: (iComputation_Parameter) casebase y query -heredados.

Conocimiento: (iComputation_Knowledge) el concepto tipo de caso al que se refiere la base de casos tiene una medida de similitud (instancia de SimilarityMeasure). No obligatorio.

Se recomienda definir instancias de SimilarityMeasure propias de la aplicación, es decir, un contexto con medidas de similitud (With_SimilarityMeasures).

Diseño: (iComputation_Design) threshold (umbral de similitud no obligatorio).

Secuencia: --

---

iRetrieve_Instance_Classification_Method: recuperación de casos por reconocimiento de instancias.

Competencia: iretrieve_task

Subtareas: irecognize_task (AssessSim_Task), iselect_case (Select_Task)

Salida: (iRetrieval_Output) atributo retrieved_cases (lista ordenada de uno o más casos recuperados)

Aplicabilidad: (iClassification_Application)

case_base - heredado

taxonomía conceptual suficientemente poblada en la zona de clasificación de las instancias.

Entrada: (iClassification_Inputs)

Paramétricos: (iClassification_Parameter) atributos casebase y query -heredados.

Conocimiento: (iClassification_Knowledge) anotaciones GTO_Specification en los conceptos para indicar cómo generalizar en cada zona de la base de conocimiento.

Diseño: (iClassification_Design)

atributo gto_specification: anotaciones en los conceptos de clasificación (instancias de GTO_specification) que indican cómo generalizar - No obligatorio.

atributo relation_path: Cadena de relaciones para especificar que lo que queremos clasificar es alguna de las componentes del individuo consulta. Si no se especifica o es vacía, se clasifica el propio individuo consulta.

atributo generalize_first, (valores posibles relation, concept, alternate_relation_first, alternate_concept_first): configura cómo relajar la clasificación.

Secuencia: --

---

iRetrieve_Concept_Classification_Method: recuperación de casos por clasificación de conceptos.

Competencia: iretrieve_task

Subtareas: iclassify_task (AssessSim_Task), iselect_case

Salida: (iRetrieval_Output) atributo retrieved_cases (lista ordenada de uno o más casos recuperados)

Aplicabilidad: (iClassification_Application)

case_base - heredado

Taxonomía de conceptos y relaciones del dominio suficientemente poblada en la zona del clasificación del concepto.

Entrada (iClassification_Inputs)

Paramétricos: (iClassification_Parameter) casebase y query -heredados.

Conocimiento: (iClassification_Knowledge) anotaciones GTO_Specification en los conceptos para indicar cómo generalizar en cada zona de la base de conocimiento.

Diseño: (iClassification_Design) (mismo individuo que el método anterior)

Secuencia: --

---

iUser_Retrieval_Method: recuperación de casos manual por el usuario.

Competencia: iretrieve_task (Retrieve_Task)

Subtareas: --

Salida: (iRetrieval_Output) atributo retrieved_cases (lista con el caso elegido por el usuario) Aplicabilidad: (iUser_Application) case_base - heredado

Entrada (iUserRetrieval_Inputs)

Paramétricos: (iUserRetrieval_Parameter) casebase y query -heredados.

Conocimiento: --

Diseño: --

Secuencia: --

---

iRandom_Retrieval_Method: recuperación de casos aleatoria

Competencia: iretrieve_task

Subtareas: --

Salida: (iRetrieval_Output) atributo retrieved_cases (lista con un caso seleccionado aleatoriamente)

Aplicabilidad: (iRandom_Retrieval_Application) case_base - heredado

Entrada (iRandom_Retrieval_Inputs)

Paramétricos: (iRandom_Retrieval_Parameter) casebase y query -heredados.

Conocimiento: --

Diseño: --

Secuencia: --

---

iRetrieve_GoalPre_Method: recuperación de casos por reconocimiento de instancias sobre los retículos resultante del AFC (construido por el método iFCA_GoalPre_Method)

Competencia: iretrieve_task (Retrieve_Task)

Subtareas: irecognizeGoalPre_task (AssessSim_Task), iselect_case (Select_Task)

Salida: (iRetrieval_Output) atributo retrieved_cases (lista ordenada de uno o más casos recuperados)

Aplicabilidad: (iRetrieve_GoalPre_Application)

case_base - heredado

El contexto tiene en los atributos goal-organization-structure y pre-organization el relleno FCA_lattice (lo que significa que se han construido los retículos).

Entrada (iRetrieve_GoalPre_Inputs)

Paramétricos: (iRetrieve_GoalPre_Parameter) casebase y query -heredados.

Conocimiento: --

Diseño: --

Secuencia: --

---

iRetrieve_SimilarityTerms_Method: recuperación de casos usando términos de similitud

Competencia: iretrieve_task (Retrieve_Task)

Subtareas: iFilter_Cases (ObtainCases_Task), iCompute_simTerms_Task(AssessSim_Task),

iselect_case (Select_Task)

Salida: (iRetrieval_Output) atributo retrieved_cases (lista ordenada de uno o más casos recuperados)

Aplicabilidad: (iRetrieve_SimilarityTerms_Application) case_base - heredado

Entrada (i iRetrieve_SimilarityTerms_Inputs)

Paramétricos: (iRetrieve_SimilarityTerms_Parameter) casebase y query -heredados.

Conocimiento: --

Diseño: --

Secuencia: --

---

iRetrieve_RelCriteria_Method: recuperación de casos usando criterios de relevancia

Competencia: iretrieve_task, iselect_case

Subtareas: iretrieve_relevance_task (AssessSim_Task), iselect_case (Select_Task)

Salida: (iRetrieval_Output) atributo retrieved_cases (lista ordenada de uno o más casos recuperados)

Aplicabilidad: (iRetrieve_RelCriteria_Application) case_base - heredado

Entrada (iRetrieve_RelCriteria_Inputs)

Paramétricos: (iRetrieve_RelCriteria_Parameter) casebase y query -heredados.

Conocimiento: --

Diseño: (iRetrieve_RelCriteria_Design)

relcri_list: lista de criterios de relevancia de los que elige el usuario final si el atributo used_criteria vale user.

used_criteria: atributo que determina qué criterio utilizar de la lista anterior. el valor user indica que es el usuario final el que elige qué criterio usar y el valor order indica que el método los prueba todos en orden de inserción.

Secuencia: --

---

iCase_Inspection_Method: recuperación exacta o inspección de casos por complección de consultas usando las reglas de dependencia resultantes de aplicar el AFC.

Competencia: iretrieve_task (Retrieve_Task)

Subtareas: iinspection_task (AssessSim_Task), iselect_case (Select_Task)

Salida: (iRetrieval_Output) atributo retrieved_cases (lista ordenada de uno o más casos recuperados) Aplicabilidad: (iCase_Inspection_Application)

case_base - heredado

Se ha costruido el retículo de conceptos formales (lo construye el método iFCA_Properties_Method)

Entrada (iCase_Inspection_Inputs)

Paramétricos: (iCase_Inspection_Parameter) casebase y query -heredados.

Conocimiento: --

Diseño: --

Secuencia: --

---

iFCA_Retrieval_Method: recuperación aproximada de casos basada en reconocimiento de instancias sobre el retículo de conceptos formales resultante de aplicar el AFC.

Competencia: iretrieve_task (Retrieve_Task)

Subtareas: --

Salida: (iRetrieval_Output) atributo retrieved_cases (lista ordenada de uno o más casos recuperados)

Aplicabilidad: (iFCA_Retrieval_Application) case_base - heredado

Entrada (iFCA_Retrieval_Inputs)

Paramétricos: (iFCA_Retrieval_Parameter) casebase y query -heredados.

Conocimiento: --

Diseño: --

Secuencia: --

ORGANIZACIÓN DE CASOS

---

iFCA_GoalPre_Method: computa los retículos de precondiciones y objetivos aplicando el AFC a los casos.

Competencia: iCB_Organization_task

Subtareas: --

Salida: implícita. Hace un cambio en el individuo contexto para poner en los atributos goal-organization-structure y pre-organization-structure del contexto el relleno FCA_lattice.

Aplicabilidad: Base de casos no vacía. Casos descritos con precondiciones y objetivos (el representante de la base de casos se clasifica como Case_With_Goals y Case_With_Pre).

Profundidad media bajo los conceptos Goal y Precondition.

Entrada: (iFCA_Goal_Pre_Input)

Paramétricos: --

Conocimiento: (iFCA_Goal_Pre_Knowledge) profundidad media bajo el concepto Goal y Precondition (no obligatorio).

Diseño: --

Secuencia: --

---

iFCA_Properties_Method: computa como estructura de organización de casos el retículo de conceptos formales y extrae las reglas de dependencia aplicando el AFC a los casos de la base.

Competencia: iCB_Organization_task

Subtareas: --

Salida: implícita. Hace un cambio en el individuo contexto para poner en los atributos case-organization-structure del contexto el relleno FCA_lattice.

Aplicabilidad: Base de casos no vacía.

Entrada: (iFCA_Properties_Input)

Paramétricos: --

Conocimiento: --

Diseño: (iFCA_Properties_Design) atributo relation_path mediante el cual el diseñador indica los atributos que se usarán para definir el contexto formal.

Secuencia: --

MÉTODOS QUE RESUELVEN LAS SUBTAREAS DERIVADAS DE LOS MÉTODOS DE RECUPERACIÓN

---

iobtain_case_base: recupera y devuelve todas las instancias de la base de casos (tipo de casos) que recibe como parámetro.

Competencia: iobtain_cases (ObtainCases_Task), iFilter_Cases_Task (ObtainCases_Task)

Subtareas: --

Salida: (iobtain_cases_output) atributo caseList que lo relaciona con el conjunto de casos que son instancias de la base de casos (representada por el parámetro case base).

Aplicabilidad: --

Entrada: (iobtain_cases_input)

Paramétricos: (iobtain_cases_parameter) casebase (heredado)

Conocimiento: --

Diseño: --

Secuencia: --

VALORACIÓN DE LA SIMILITUD

---

iNumeric_Simcomputation_Method: computa la similitud entre la consulta (parámetro query) y cada uno de los casos del conjunto dado por el requisito de secuencia caseList si existe y si no de todos los de la base de casos especificada en el parámetro casebase.

Competencia: icompute_similarity(AssessSim_Task), iFilter_Cases_Task (ObtainCases_Task)

Subtareas: --

Salida: (icompute_similarity_output)

atributo caseList que lo relaciona con el conjunto de casos de salida.

atributo simValues que lo relaciona con una lista ordenada de pares (Ci, SVi), donde Ci es el nombre de un caso y SVi el valor de similitud entre el caso Ci y la consulta.

Aplicabilidad: --

Entrada (iNumeric_Simcomputation_Input):

Paramétricos: (iComputation_Parameter) casebase y query (heredados)

Conocimiento: (iComputation_Knowledge) definir Similarity Measures (heredado)

Diseño: --

Secuencia: (iNumeric_Simcomputation_Sequence) atributo caseList.

---

irecognizeGoal_Method crea y clasifica un individuo con los objetivos de la consulta para recuperar las instancias similares según los conceptos del retículos de objetivos bajo los que se haya clasificado.

Competencia: irecognizeGoalPre_task (AssessSim_Task)

Subtareas: --

Salida: (irecognize_output) atributo caseList. (mismo individuo que el método de reconocimiento de instancias).

Aplicabilidad: (irecognizeGoal_Application) El retículo de objetivos se ha computado previamente (por el método iFCA_GoalPre_Method).

Entrada: (iRecognizeGoal_Input)

Paramétricos: (iRecognizeGoal_Method) query - heredado.

Conocimiento: (iRecognizeGoal_Knowledge) Contexto con retículo de objetivos.

Diseño: --

Secuencia: --

---

irecognizePre_Method crea y clasifica un individuo con la situación descrita por la consulta para recuperar las instancias similares según los conceptos del retículos de precondiciones bajo los que se haya clasificado.

Competencia: irecognizeGoalPre_task (AssessSim_Task)

Subtareas: --

Aplicabilidad: (irecognizePre_Method) El retículo de precondiciones se ha computado previamente (por el método iFCA_GoalPre_Method)

Salida: (irecognize_output) atributo caseList. (mismo individuo que el método de reconocimiento de instancias).

Entrada: (iRecognizePre_Input)

Paramétricos: (iRecognizePre_Method) query - heredado.

Conocimiento: (iRecognizePre_Knowledge) Contexto con retículo de precondiciones.

Diseño: --

Secuencia:--

---

irecognizeGoalPre_Method crea y clasifica un individuo con la situación descrita por la consulta en el retículo de precondiciones y crea y clasifica otro individuo en el retículo de objetivos y combina los resultados.

Competencia: irecognizeGoalPre_task (AssessSim_Task)

Subtareas: --

Salida: (irecognize_output) atributo caseList. (mismo individuo que el método de reconocimiento de instancias).

Aplicabilidad: (irecognizeGoalPre_Application) Contexto en que ambos retículos el de precondiciones y el de objetivos se han computado previamente (los computa el método iFCA_GoalPre_Method)

Entrada: (irecognizeGoalPre_Input)

Paramétricos: (irecognizeGoalPre_Parameter) query-heredado.

Conocimiento: (irecognizeGoalPre_Knowledge) Contexto con retículos de precondiciones y objetivos.

Diseño: --

Secuencia:--

---

iinspection_method: completa la consulta usando las reglas de dependencia extraídas del AFC y la clasifica en el retículo para recuperar casos cercanos.

Competencia: iinspection_task (AssessSim_Task)

Subtareas: iquery_completion_task, iquery_FCA_retrieval_task

Salida: (iinspection_output) atributo caseList.

Aplicabilidad (iinspection_application): se ha construido el retículo AFC de propiedades (lo construye el método iFCA_Properties_Method) y se han extraido las reglas de dependencias (Dependence_Rule_Concept)

Entrada: (iinspection_input)

Paramétricos: (iinspection_parameter) query - heredado

Conocimiento: (iinspection_knowledge) se ha construido el retículo AFC y las reglas de dependencia.

Diseño: --

Secuencia: --

---

iquery_completion_method método de resolución interactivo que se encarga de construir y completar la consulta del usuario usando las reglas de dependencia extraídas de la aplicación del AFC a los casos.

Competencia: iquery_completion_task

Subtareas: --

Salida: (iquery_completion_output) atributo completed-query

Aplicabilidad: (iinspection_application) se ha construido el retículo AFC.

Entrada: (iinspection_input)

Paramétricos: (iinspection_parameter) query - heredado

Conocimiento: (iinspection_knowledge) se ha construido el retículo AFC y las reglas de dependencia.

Diseño: --

Secuencia: --

---

iquery_FCA_retrieval_method método de resolución que se encarga de recuperar los individuos clasificados bajo los conceptos más específicos del retículo AFC bajo los que se clasifica el individuo consulta.

Competencia: iquery_FCA_retrieval_task

Subtareas: --

Salida: (iinspection_output) atributo caseList.

Aplicabilidad: (iinspection_application) se ha construido el retículo AFC.

Entrada: (iquery_FCA_retrieval_input)

Paramétricos: --

Conocimiento: (iinspection_application) se ha construido el retículo AFC.

Diseño: --

Secuencia: (iquery_FCA_retrieval_sequence) atributo completed-query

---

iretrieve_relevance_method: valora la similitud de los casos utilizando el intérprete de consultas de Loom. Para ello genera una consulta utilizando el criterio de relevancia que recibe como parámetro.

Competencia: iretrieve_relevance_task (AssessSim_Task), iFilter_Cases (ObtainCases_Task)

Subtareas: --

Salida: (iretrieve_relevance_output) atributo caseList

Aplicabilidad: --

Entrada: (iretrieve_relevance_input)

Paramétricos: (iretrieve_relevance_parameter)

atributos query y casebase -heredados

atributo relevance_criteria - añadido

Conocimiento: (iretrieve_relevance_knowledge) existen subrelationes de relevanceCriteria que representan los criterios de relevancia predefinidos (recomendable).

Diseño: --

Secuencia: --

---

iinstance_classification_and: recupera aquellos individuos que se clasifican exactamente igual que los individuos del final de la cadena relation_path (a partir de la consulta - query).

Competencia: irecognize_task (AssessSim_Task)

Subtareas: --

Salida: (irecognize_output) atributo caseList

Aplicabilidad: (iClassification_Application) -heredado.

Entrada: (iClassification_Inputs) -heredado

Paramétricos: (iClassification_Parameter) atributos casebase y query -heredados.

Conocimiento: (iClassification_Knowledge) anotaciones GTO_Specification en los conceptos para indicar cómo generalizar en cada zona de la base de conocimiento.

Diseño: (iClassification_Design) -heredado (mismo individuo que iRetrieve_Instance_Classification_Method)

atributo gto_specification: anotaciones en los conceptos de clasificación (instancias de GTO_specification) que indican cómo generalizar - No obligatorio.

atributo relation_path: Cadena de relaciones para especificar que lo que queremos clasificar es alguna de las componentes del individuo consulta. Si no se especifica o es vacía, se clasifica el propio individuo consulta.

atributo generalize_first, (valores posibles relation, concept, alternate_relation_first, alternate_concept_first): configura cómo relajar la clasificación.

Secuencia: --

---

iinstance_classification_or: recupera aquellos individuos cuya clasificación comparte al menos un concepto con los individuos del final de la cadena relation_path (a partir de query).

Competencia: irecognize_task (AssessSim_Task)

Subtareas: --

Salida: (irecognize_output) atributo caseList

Aplicabilidad: (iClassification_Application) -heredado.

Entrada: (iClassification_Inputs) -heredado

Paramétricos: (iClassification_Parameter) atributos casebase y query -heredados.

Conocimiento: (iClassification_Knowledge) anotaciones GTO_Specification en los conceptos para indicar cómo generalizar en cada zona de la base de conocimiento.

Diseño: (iClassification_Design) -heredado (mismo individuo que iRetrieve_Instance_Classification_Method)

Secuencia: --

---

iclassify_method: recupera como individuos similares a la consulta aquellos individuos que son instancia del concepto que representa la consulta.

Competencia: iclassify_task (AssessSim_Task)

Subtareas: --

Salida: (iclassify_output) atributo caseList

(iClassification_Application) -heredado.

Entrada: (iClassification_Inputs) -heredado

Paramétricos: (iClassification_Parameter) atributos casebase y query -heredados.

atributo relation_path: Cadena de relaciones.

Conocimiento: --

Diseño: (iClassification_Design)

taxonomía conceptual suficientemente poblada en la zona de clasificación de las instancias.

anotaciones en los conceptos de clasificación (instancias de GTO_specification) que indican cómo generalizar - No obligatorio.

Secuencia: --

---

iCompute_simTerms_method se encarga de computar los términos de similitud entre la consulta y cada uno de los casos en la lista caseList (requisito de secuencia).

Competencia: iCompute_simTerms_task (AssessSim_Task)

Subtareas: --

Salida: (iCompute_simTerms_output) atributo caseList

Aplicabilidad: --

Entrada: (iCompute_simTerms_output)

Paramétricos: (iCompute_simTerms_parameter) atributos query y casebase -heredados

Conocimiento: --

Diseño: --

Secuencia: atributo caseList.

SELECCIÓN DE CASOS

---

iuserselectcase_method: selección de casos por el usuario

Competencia: iselect_best (Select_Task), iselect_case (Select_Task)

Subtareas: --

Salida: (iRetrieval_Output) atributo retrieved_cases (lista ordenada de uno o más casos recuperados). Salida compartida con los métodos de recuperación por ser un método asociado a la última subtarea de las secuencias de tareas derivadas por los métodos de recuperación.

Aplicabilidad: --

Entrada (iuserselectcase_input)

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia: (iuserselectcase_sequence) atributo caseList con la lista de casos candidatos.

---

iselectallcases_method: selecciona todos los casos con un valor de similitud superior al valor umbral si existe, y si no todos los de la lista que recibe en el atributo caseList.

Competencia: iselect_best (Select_Task), iselect_case (Select_Task)

Subtareas: --

Salida:(iRetrieval_Output) atributo retrieved_cases (lista ordenada de uno o más casos recuperados).

Aplicabilidad:--

Entrada (iselectallcases_input)

Paramétricos: -

Conocimiento: -

Diseño: (iselectallcases_design) atributo threshold.- heredado.

Secuencia: (iselectallcases_sequence) atributo caseList con la lista de casos. Además, si existe el atributo simValues con una lista de pares (caso, valor de similitud) lo utiliza.

---

iselectmaxcase_method: selecciona el primer caso de la lista

Competencia: iselect_best (Select_Task), iselect_case (Select_Task)

Subtareas: --

Salida:(iRetrieval_Output) atributo retrieved_cases (lista ordenada de uno o más casos recuperados).

Aplicabilidad: --

Entrada: (iselectmaxcase_input)

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia: (iselectmaxcase_sequence) atributo caseList con el conjunto de casos recuperados, aunque si existe simValues con la lista de pares (caso, valor de similitud) la utiliza.

---

iselect_computation_method: selecciona el caso mejor aplicando el método de cómputo de similitud entre la consulta (query) y los casos resultantes de la tarea de valoración de similitud anterior (atributo caseList).

Competencia: iselect_case (Select_Task)

Subtareas: icompute_similarity (AssessSim_Task), iselect_best (Select_Task)

Salida:(iRetrieval_Output) atributo retrieved_cases (lista ordenada de uno o más casos recuperados).

Aplicabilidad: --

Entrada: (iselect_computation_input)

Paramétricos: parámetro query - heredado

Conocimiento: Contexto con medidas de similitud (With_SimilarityMeasures).

Diseño: (iselect_computation_design) threshold (umbral de similitud).

Secuencia: (iselect_computation_sequence) atributo caseList.

---

iselect_RelCriteria_Method: selecciona el caso mejor aplicando el método de los criterios de relevancia. Selecciona de casos resultantes de la tarea de valoración de similitud anterior (atributo caseList) aquellos que cumplan los criterios de relevancia especificados en el método

Competencia: iselect_candidates [ISIA1](Select_Task)

Subtareas: iretrieve_relevance_task (AssessSim_Task), iselect_case [ISIA2](Select_Task)

Salida:(iRetrieval_Output) atributo retrieved_cases (lista ordenada de uno o más casos recuperados).

Aplicabilidad: --

Entrada: (iselect_RelCriteria_Inputs)

Paramétricos: parámetro query - heredado

Conocimiento: --

Diseño: (iselect_RelCriteria_design)

relcri_list: lista de criterios de relevancia de los que elige el usuario final si el atributo used_criteria vale user.

used_criteria: atributo que determina qué criterio utilizar de la lista anterior. el valor user indica que es el usuario final el que elige qué criterio usar y el valor order indica que el método los prueba todos en orden de inserción.

Secuencia: (iselect_relcriteria_sequence) atributo caseList.

ADAPTACIÓN

---

ireuse_case_method adapta el primero de los casos recuperados en la tarea de recuperación de casos.

Competencia: ireuse_task

Subtareas: icopy_solution_task (Copy_Solution_Task),

iadapt_solution_task (Adapt_Solution_Task)

Salida: (iReuse_Output) atributos reuse-result (success o failure) y adaptedCase

Aplicabilidad: (iReuse_Application) requiere la adaptación de casos con solución, el relleno de casebase debe ser clasificado como CaseBase_with_Solution.

Entrada: ( )

Paramétricos: query (heredado)

Conocimiento: -

Diseño:

Secuencia: atributo retrieved_cases con la lista de casos recuperados de los que adaptará el primero.

MÉTODOS QUE RESUELVEN LAS SUBTAREAS DERIVADAS DE LOS MÉTODOS DE ADAPTACIÓN

---

icopy_solution_method hace una copia de la solución del caso a adaptar como la solución de la consulta, de forma que el caso original no se modifica.

Compencia: icopy_solution_task (Copy_Solution_Task)

Subtareas: --

Salida: (iCopy_Solution_output) atributo casetoAdapt (el relleno será instancia de Case_with_Solution) individuo que incluye la descripción de la consulta y la solución del caso recuperado, atributo caseList con la lista de casos recuperados.

Aplicabilidad (iReuse_Application) base de casos con solución (heredado).

Entrada: (icopy_solution_input)

Paramétricos: (icopy_solution_parameter) atributo query (heredado)

Conocimiento: --

Diseño: --

Secuencia: atributo caseList con la lista de casos recuperados.

---

iAdapt_by_Substitution_Method se basa en adaptar la solución del caso recuperado (requisito de secuencia casetoAdapt) realizando sustituciones genéricas guiadas por el conocimiento del dominio

Competencia: iadapt_solution_task (Adapt_Solution_Task)

Subtareas: (cycle) isubstitution_strategy_task (Select_Strategy_Task),

ifind_adaptation_actions_task (Select_Discrepancy_Task),

imodify_solution_task (Modify_Solution_Task)

Salida: (iReuse_Output) atributos reuse-result (success o failure) y adaptedCase

Aplicabilidad: (iReuse_Application) base de casos con solución (heredado)

Entrada (iAdapt_by_Substitution_Inputs)

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia (iAdapt_Solution_Sequence): atributo casetoAdapt, atributo caseList.

---

iAdapt_Specialized_Method adapta la solución del caso recuperado (requisito de secuencia casetoAdapt) utilizando estrategias de adaptación especificadas por el diseñador.

Competencia: iadapt_solution_task (Adapt_Solution_Task)

Subtareas: (cycle) iretrieve_adaptation_strategy_task (Select_Strategy_Task),

ifind_adaptation_actions_task (Select_Discrepancy_Task),

imodify_solution_task (Modify_Solution_Task)

Salida: (iReuse_Output) atributos reuse-result (success o failure) y adaptedCase

Aplicabilidad: (iAdapt_Specialized_Application) base de casos con solución (heredado), se han definido estrategias de adaptación y tipos de problemas, se comprueba si existen instancias de Adaptation_Strategy y subconceptos de Problem_Type respectivamente (requisito de conocimiento obligatorio).

Entrada (iAdapt_Specialized_Input)

Paramétricos: --

Conocimiento: (iAdapt_Specialized_Knowledge) existen instancias de Adaptation_Strategy y subconceptos de Problem_Type.

Diseño: --

Secuencia (iAdapt_Solution_Sequence): atributo casetoAdapt, atributo caseList.

---

iretrieve_adaptation_strategy_method:identifica los problemas que plantea el caso recuperado en la situación actual y en base a ellos, recupera las estrategias especificadas por el diseñador.

Competencia: iretrieve_adaptation_strategy_task

Subtareas: --

Salida: (iadaptation_strategy_output) atributo adaptation_strategy que hace referencia a la estrategia recuperada a aplicar en las siguientes subtareas de la secuencia.

Aplicabilidad: (iAdapt_Specialized_Application) heredado.

Entrada (iAdapt_Specialized_Input) heredado

Paramétricos: --

Conocimiento: (iAdapt_Specialized_Knowledge) existen instancias de Adaptation_Strategy y subconceptos de Problem_Type.

Diseño: --

Secuencia: --

---

isubstitution_strategy_method: recupera una estrategia genérica de sustitución de elementos que está predefinida que se puede configurar con los requisitos de diseño.

Competencia: isubstitution_strategy_task

Subtareas: --

Salida: (iadaptation_strategy_output) atributo adaptation_strategy

Aplicabilidad:

Entrada (isubstitution_strategy_inputs):

Paramétricos: --

Conocimiento: --

Diseño: (isubstitution_strategy_design):

atributo include-transformation cuyo relleno es una instancia del concepto Transformation.

atributo delete_transformation, para eliminar una transformación existente, el relleno es un nombre de una transformación de las existentes en la estrategia. (las características de una estrategia de adaptación, incluyendo sus transformaciones, se pueden ver usando la función de la API view_adaptation_strategy).

Secuencia: --

---

iuser_find_adaptation_actions_Method se basa en la interacción con el usuario final

Competencia: ifind_adaptable_actions_task

Subtareas: --

Salida: (ifind_adaptable_actions_output) atributo action-list con la lista de acciones (instancias de Transformation_Action) que describen qué transformaciones realizar y sobre qué elementos. También se mantiene la referencia al atributo adaptation_strategy.

Aplicabilidad:

Entrada:

Paramétricos: --

Conocimiento: --

Diseño:

Secuencia: --

---

ifix_adaptation_items_Method recupera los elementos que cumplen una lista de comprobación (requisito de diseño)

Competencia: ifind_adaptable_actions_task

Subtareas: --

Salida: (ifind_adaptable_actions_output) atributo action-list con la lista de acciones (instancias de Transformation_Action) que describen qué transformaciones realizar y sobre qué elementos. También se mantiene la referencia al atributo adaptation_strategy.

Aplicabilidad:

Entrada: (ifix_adaptation_items_Input)

Paramétricos: (ifix_adaptation_items_Parameter)

atributo toadapt: especificación de una lista de elementos a adaptar. Deben ser compatibles con la lista de comprobación dada como requisito de diseño.

Conocimiento: --

Diseño: (ifix_adaptation_items_Design)

atributo toadapt: especificación de una lista de comprobación que determina los elementos a adaptar.El formato para especificar los individuos elegidos es una cadena (l1, .., ln) donde cada li es de la forma: (relación restricción) o (relación) y restricción es un concepto compatible con el rango de la relación.

atributo filter: nombre de una función que permite ordenar los candidatos.La función Lisp debe recibir y devolver la lista de candidatos.

Secuencia (ifind_adaptable_actions_sequence): atributo adaptation_strategy

---

isystem_find_adaptable_actions_method utiliza las diferencias entre los valores de los atributos de las descripciones del caso y la consulta, y la representación explícita de cómo los elementos de la solución se ven afectados por estas diferencias (depends_on).

Competencia: ifind_adaptable_actions_task

Subtareas: --

Salida: (ifind_adaptable_actions_output) atributo action-list con la lista de acciones (instancias de Transformation_Action) que describen qué transformaciones realizar y sobre qué elementos. También se mantiene la referencia al atributo adaptation_strategy.

Aplicabilidad:

Entrada:

Paramétricos: --

Conocimiento: (isystem_find_adaptation_actions_knowledge) se usa la relación depends_on para marcar las dependencias entre ciertos elementos del dominio.

Diseño: (isystem_find_adaptation_actions_design)

atributo select_transformation_by cuyo relleno es una de las instancias de Select_Transformation: random, user o use-weight.

Secuencia (ifind_adaptable_actions_sequence): atributo adaptation_strategy

---

imodify_solution_method se encarga de aplicar las transformaciones indicadas por la estrategia de adaptación (requisito de secuencia)

Competencia: imodify_solution_task

Subtareas: (cycle) iapply_transformation_task (Apply_Transformation_Task),

ilocal_revisión_task (Local_Revision_Task)

Salida: (iReuse_Output) atributos reuse-result (success o failure) y adaptedCase. También se mantiene la referencia al atributo adaptation_strategy con las estrategias utilizadas.

Aplicabilidad: --

Entrada (iModify_Solution_Inputs)

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia (imodify_solution_sequence): atributo case_to_Adapt, atributo caseList con la lista de casos recuperados, atributo action-list con la lista de acciones (instancias de Transformation_Action)

---

iapply_transformation_method aplicar una transformación sobre un elemento del caso.

Competencia: iapply_transformation_task

Subtareas: --

Salida: (iReuse_Output) atributos reuse-result (success o failure) y adaptedCase. También se mantiene la referencia al atributo adaptation_strategy con las estrategias utilizadas.

Aplicabilidad: --

Entrada (iApply_Transformation_Inputs)

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia: (iapply_transformation_sequence) atributo action cuyo relleno es una instancia de Transformation_Action, que hace referencia a la transformación (transformation) a realizar sobre un elemento (item).

MÉTODOS DE BÚSQUEDA DE SUSTITUTOS <Search-Substitutes-Method instance>. Se corresponden con los métodos de recuperación de casos y no se usan para resolver una tarea de una secuencia sino que describe una estrategia de adaptación y se aplica al aplicar una transformación de sustitución que haga referencia a él.

---

iSearch_computational_method: búsqueda de sustitutos por cómputo de similitud numérica.

iSearch_Instance_Classification_Method: búsqueda de sustitutos por reconocimiento de instancias.

iSearch_concept_classification_method: búsqueda de sustitutos por clasificación de conceptos.

iUser_Search_Method: búsqueda de sustitutos manual por el usuario.

iRandom_Retrieval_Method: búsqueda de sustitutos aleatoria.

iRetrieve_SimilarityTerms_Method: búsqueda de sustitutos usando términos de similitud

iRetrieve_RelCriteria_Method: búsqueda de sustitutos usando criterios de relevancia

---

iuser_local_revisión_method delega al usuario la validación de cada paso de transformación.
Competencia: iapply_transformation_task

Subtareas: --

Salida: (iReuse_Output) atributos reuse-result (success o failure) y adaptedCase.

Aplicabilidad: --

Entrada (iLocal_Revision_Inputs)

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia (iLocal_Revision_Sequence) atributo adaptedCase.

REVISIÓN

---

iRevise_Method: revisa el caso adaptado que resulta de la tarea de adaptación.

Competencia: iRevise_Task

Subtareas: iSystem_Revision_task (System_Revision_Task),

iUser_Revision_task (User_Revision_Task)

Salida (iRevise_Output) atributos revise-result (success o failure) y revisedCase que es el caso en su estado final, pero si result es fallo no satisface los requisitos de corrección establecidos.

Aplicabilidad: --

Entrada (iRevise_Inputs)

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia: (iRevise_Sequence) atributo adaptedCase

---

iSystem_Revision_Method: método de revisión automático del caso adaptado (adaptedCase)

Competencia: iSystem_Revision_task (System_Revision_Task)

Subtareas: iSystem_Evaluation_Task (Evaluation_Task), isystem_repair_task (Repair_Task)

Salida (iRevise_Output) atributos revise-result (success o failure) y revisedCase que es el caso en su estado final, pero si result es fallo no satisface los requisitos de corrección establecidos.

Aplicabilidad: (iRevise_Application) se requiere la definición de tipos de fallo (instancias de Fail_Type) y estrategias de reparación (instancias de Repair_Strategy).

Entrada (iRevise_input)

Paramétricos: --

Conocimiento: (iRevise_knowledge)

se requiere la definición de tipos de fallo (instancias de Fail_Type) y estrategias de reparación (instancias de Repair_Strategy).

Diseño: --

Secuencia: (iRevise_Sequence) atributo adaptedCase

---

iUser_Revision_Method: revisión manual (usuario) del caso adaptado.

Competencia: iUser_Revision_task (User_Revision_Task)

Subtareas: --

Salida (iRevise_Output) atributos revise-result (success o failure) y revisedCase que es el caso en su estado final, pero si result es fallo no satisface los requisitos de corrección establecidos.

Aplicabilidad: --

Entrada (iuser_revision_input)

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia: (iRevise_Sequence) atributo adaptedCase

---

iSystem_Evaluation_Method: evaluación automática de los fallos del caso adaptado

Competencia: iSystem_Evaluation_Task (Evaluation_Task)

Subtareas: --

Salida (iSystem_Evaluation_Method)

Atributo fail_list. Su relleno es una lista de conceptos que representan los tipos de fallo de reparación que tiene el caso adaptado.

Aplicabilidad: (iRevise_Application) se requiere la definición de tipos de fallo (instancias de Fail_Type) y estrategias de reparación (instancias de Repair_Strategy).

Entrada (iRevise_input)

Paramétricos: --

Conocimiento: (iRevise_knowledge) se requiere la definición de tipos de fallo (instancias de Fail_Type) y estrategias de reparación (instancias de Repair_Strategy).

Diseño: --

Secuencia: (iRevise_Sequence) atributo adaptedCase

---

iSystem_Repair_Method: repara los fallos del caso adaptado aplicando la estrategia de reparación (requisito de secuencia - repair strategy).

Competencia: iSystem_Repair_Task (Repair_Task)

Subtareas: (cycle) iselect_repair_strategy_task (Select_Repair_Strategy_Task),

iapply_repair_strategy_task (Apply_Repair_Strategy_Task).

Salida (iRevise_Output) atributos revise-result (success o failure) y revisedCase que es el caso en su estado final, pero si result es fallo no satisface los requisitos de corrección establecidos.

Aplicabilidad: --

Entrada (iSystem_Repair_Inputs)

Paramétricos: --

Conocimiento: --

Diseño: ---

Secuencia: (iSystem_Repair_Sequence) atributo fail_list

---

iselect_repair_strategy_method: selecciona la estrategia de reparación a utilizar

Competencia: iselect_repair_strategy_task (Select_Repair_Strategy_Task)

Subtareas: --

Salida (iselect_repair_strategy_output) atributo repair_strategy (su relleno es instancia de Repair_Strategy).

Aplicabilidad: --

Entrada (iselect_repair_strategy_input)

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia: (iselect_repair_strategy_sequence)

atributo fail_list que contiene la lista de fallos de reparación que dan acceso la estrategia de reparación.

---

iapply_repair_strategy_method: aplica la estrategia de reparación que recibe del método anterior.

Competencia: iapply_repair_strategy_task (Apply_Repair_Strategy_Task).

Subtareas: --

Salida (iRevise_Output) atributos revise-result (success o failure) y revisedCase que es el caso en su estado final, pero si result es fallo no satisface los requisitos de corrección establecidos. También el atributo repair_strategy que hace referencia a las estrategias utilizadas.

Aplicabilidad: --

Entrada (iapply_repair_strategy_input)

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia: (iapply_repair_strategy_sequence) existe el atributo repair_strategy, que contiene la estrategia de reparación.

APRENDIZAJE

---

iRetain_Method: aprendizaje de conocimiento tras la resolución del problema.

Competencia: iRetain_Task

Subtareas: iretain_case_method

Salida: (iRetain_Output) atributo retainedCase (cuando se aprenda un caso).

Aplicabilidad: --

Entrada: --

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia: --

---

iretain_case_method: aprendizaje de un nuevo caso que se incorporará a una base de casos temporal hasta que en la resolución de la tarea de mantenimiento se decida incorporarlo definitivamente a la base de casos.

Competencia: iretain_case_task

Subtareas: --

Salida: (iRetain_Output) atributo retainedCase.

Aplicabilidad: --

Entrada: --

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia: existencia de los atributos revise-result y revisedCase que provienen de la tarea de revisión. Es decir, sólo se pueden aprender casos que estén revisados, bien automáticamente o bien por el usuario final.

---

iretain_retrieval_knowlege_method: aprendizaje de conocimiento de recuperación en forma de resultado de los casos.

Competencia: iretain_retrieval_knowlege_task

Subtareas: --

Salida: (iRetain_Output)

Aplicabilidad: --

Entrada: (iretain_retrieval_knowlege_input)

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia: (iretain_retrieval_knowlege_sequence) existencia de los atributos retrieved_case, y el atributo revise-result cuyo valor determina el resultado que se aprende (éxito o fallo).

---

iretain_reuse_knowledge_method: aprendizaje de conocimiento de adaptación en forma de pesos en las estrategias utilizadas.

Competencia: iretain_reuse_knowledge_task

Subtareas: --

Salida: (iRetain_Output)

Aplicabilidad: --

Entrada: (iretain_reuse_knowledge_input)

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia: (iretain_reuse_knowledge_sequence) existencia de los atributos adaptation_strategy y revise-result cuyo valor determina si el aprendizaje es positivo (sumamos 1 al peso de la estrategia) o negativo (resta 1).

---

iretain_revise_knowledge_method: aprendizaje de conocimiento de revisión en forma de pesos en las estrategias de reparación utilizadas.

Competencia: iretain_revise_knowlege_task

Subtareas: --

Salida: (iRetain_Output)

Aplicabilidad: --

Entrada: (iretain_revise_knowledge_input)

Paramétricos: --

Conocimiento: --

Diseño: --

Secuencia: (iretain_revise_knowledge_sequence) existencia de los atributos repair_strategy y el atributo revise-result cuyo valor determina el tipo de aprendizaje (positivo o negativo).

APÉNDICE D: FUNCIONES DE LA API DE COLIBRI

1. Modulo de visualización y modelado del dominio

```
(defun import-definitions (file-name))
    Función que recibe el nombre de archivo Loom y lo incorpora a la base de
    conocimiento actual. Se usa para importar distintas ontologías, cada una de ellas
    está guardada en un archivo Loom.
    Ejemplo de llamada: (import-definitions
    "/export/home/usr/belen/loom/COLIBRI/CBROnto/cbronto.lisp")
(defun view-cbronto-concepts () )
(defun view-cbronto-relations () )
    Funciones que imprimen (interfaz textual) y devuelven (interfaz gráfica) la
    taxonomía de conceptos y relaciones de CBROnto, respectivamente.
(defun view-terms (root))
    Imprime (interfaz textual) y devuelve (interfaz gráfica) todos los terminos
    clasificados bajo el termino (concepto o relación) root dado
    Ejemplo de llamada (view-terms 'domain-concept)
(defun view-terms-one-level (root))
    Igual que la anterior pero sólo baja un nivel debajo del termino root
    Ejemplo de llamada (view-terms 'domain-concept)
(defun list-case-types (&key (user nil))
    Imprime (interfaz textual) y devuelve (interfaz gráfica) los tipos de casos
    existentes. El parámetro user permite distinguir entre los predefinidos y los
    definidos por el usuario.
(defun how-many-cases (case-type))
    Imprime (interfaz textual) y devuelve (interfaz gráfica) el número de casos del tipo
    dado que hay en la base. Usado con CASE obtiene el número total de casos de todos
    los tipos.
(defun list-case-base (case-type))
    Imprime (interfaz textual) y devuelve (interfaz gráfica) los casos del tipo dado que
    hay en la base. Usado con CASE obtiene todos los casos de todos los tipos.
(defun get-documentation (objeto))
    Imprime (interfaz textual) y devuelve (interfaz gráfica) la documentación asociada
    al objeto dado, que puede ser un concepto, individuo o relación.
    Ejemplo de llamada:  (get-documentation 'CurrentMeasure)
                          (get-documentation 'tipoCoche)
(defun put-documentation (objeto documentation))
    añade al objeto la documentación textual.
(defun get-documentation-list (list))
    Imprime (interfaz textual) y devuelve (interfaz gráfica)la documentación de una
    lista de objetos
    Ejemplo de llamada:  (get-documentation-list (get-subconcepts (fc caseType)))
(defun print-concept (concept))
    Función que imprime (interfaz textual) y devuelve (interfaz gráfica) todas las
    características y restricciones de la definición del concepto dado.
(defun print-individual (individual &key (recursively nil)))
```

Función que imprime (interfaz textual) y devuelve (interfaz gráfica) las características de un individuo . El parámetro permite controlar si se muestran tambien los valores de forma recursiva hasta los elementos terminales de la estructura de representación.

(defun define-concept ())

Función que define un concepto. Hace una llamada a la función defconcept de Loom

(defun define-relation ())

Función que define una relación. Hace una llamada a la función defrelation de Loom

(defun define-individual ())

Función que define un individuo. Hace una llamada a la función tell de Loom

(defun obtener-instancias (C &key (direct nil)))

Función que obtiene las instancias (directas o no) de un concepto. Hace una llamada a la función get-instances de Loom

(defun GTO (C &key (level 0) (direct t)))

El operador GTO (Generic Travel Operator) toma como punto de partida un concepto C y el nivel al que queremos recuperar, teniendo en cuenta que los niveles positivos significan caminos descendentes en la jerarquía y los negativos significan caminos ascendentes a partir de C. El parámetro direct indica si el operador debe recuperar todas las instancias de los conceptos considerados o sólo las instancias directas,

(defun get-concepts (C &key (level 1)))

Función que obtiene los subconceptos (hasta el nivel dado) de un concepto, donde 0 sólo devolvería el propio concepto, 1 los subconceptos directos, -1 los superconceptos directos, ...

Se implementa mediante llamadas a las funciones get-subconcepts y get-superconcepts de Loom

(defun get-relations (C &key (level 1)))

Igual para relaciones.

(defun add-suconcept (C1 Cup))

Añade el concepto Cup a los superconceptos de C1, es decir, hace que C1 pase a ser subconcepto de Cup.

Ejemplo de llamada: (add-concept-type 'cars 'case-description)

(defun add-subrelation (R1 Rup))

Hace que la relación R1 sea subrelación de Rup

Ejemplo de llamada: (add-relation-type 'color 'descriptive-property)

ASERTOS EN INDIVIDUOS

(defun add-type (i1 C1))

Hace que el individuo i1 sea instancia directa del concepto C1.

(defun add-descriptor (C1 R card))

Añade una relación como descriptor de un cierto concepto. Se pueden incluir restricciones de cardinalidad.

(defun add-role-filler (i1 R1 V1))

Hace que el individuo i1 tenga el valor o individuo V1 como relleno del atributo (relación) R1.

(defun add-role-restriction (i1 R1 C1))

Restringe en el individuo i1 que todos sus rellenos en el atributo (relación) R1 sean instancias del concepto C1.

(defun drop-object (o))

Elimina el objeto (concepto, relación o individuo) cuyo nombre recibe como parámetro.

(defun concept-restrictions (c))

```
    Funcion que obtiene las restricciones propias y heredadas en un concepto. Devuelve
    una lista de asociación con las relaciones-roles de un concepto:
    "relacion" ---> objeto relacion de LOOM
    "min" ---> cardinalidad minima de la relacion
    "max" ---> cardinalidad maxima de la relacion
    "tipos" ---> tipo (s) o concepto (s) que pueden ser fillers del role o relació
    "valores" ---> concepto (s) o instancia (s) fillers del role o relación
    "defecto" ---> concepto (s) o instancia (s) que son fillers del role por defecto
       Los tres ultimos son, a su vez, listas.
    Esta función se utiliza para que la interfaz gráfica construya obtenga dinámicamente
    la informacion necesaria para construir el formulario de creacion de instancias de
    un concepto. Ejemplo de llamada (concept-restrictions 'tipoCoche)

FUNCIONES DE ACCESO A LOOM
(defun loom-list-knowledge-base (&optional knowledgeBase))
(defun loom-pprint-object (object))
(defun loom-object-name (object))
(defun loom-get-concept (concept))
(defun loom-get-relation (relation))
(defun loom-get-instance (instance))
(defun loom-get-instances (concept &key direct-p asserted-p))
(defun loom-get-types (instance &key direct-p asserted-p))
(defun loom-list-role-names&values (instance))
(defun subconceptos-kb (concepto &key (direct-p nil) (primitive-p nil))
(defun subrelaciones-kb (relacion &key (direct-p nil))
```

2. Modulo de definición de casos, consultas y tipos de usuario

```
(defun make-type-of-case (name type documentation))
    Funcion que hace que el tipo de casos de nombre name pase a ser de tipo type
    (cualquier subconcepto de CASE) del que heredada sus propiedades. Si el tipo de
    casos name no existe lo crea.
    Ejemplo de llamada: (make-type-of-case 'tipoCoche 'descriptive-case "el tipo de
    casos que representan coches.")
(defun create-type-of-case (name type desc sol res kor doc)
    Crea un concepto tipo de caso incluyendo informacion de su tipo de descripcion de
    solucion y de resultado (que deben ser conceptos creados previamente mediante las
    funciones make-description-type y make-result-type, respectivamente, o conceptos del
    dominio.
    Ejemplo de llamada: (create-type-of-case 'tipoCoche 'descriptive-case 'cars nil nil
    'search "el tipo de casos que representan coches.")
(defun make-description-type (name description))
    Función que crea un nuevo tipo de descripción, es decir, un subconcepto (coherente)
    del concepto primitivo Case_Result.
(defun make-result-type (name description))
    Función que crea un nuevo tipo de resultado, es decir, un subconcepto (coherente)
    del concepto primitivo Case_Result.
(defun make-solution-type (name description))
```

Función que crea un nuevo tipo de solución, es decir, un subconcepto (coherente) del concepto primitivo Case_Solution.

(defun view-type (tipoCaso))

Función que imprime (interfaz textual) y devuelve (interfaz gráfica) las restricciones de tipo de descripcion, solution y resultado asociadas a un tipo de caso. Esta función es necesaria, por ejemplo, para construir las interfaces de definición de nuevos casos de un tipo.

(defun create-case (case-name case-type documentation))

Función que crea un caso del tipo dado dado. Definimos una instancia de ese concepto.

Las carácterísticas del individuo caso se establecen con las funciones siguientes que se usan para asignar las componentes (individuos) descripción, solución, resultado, que deben estar previamente creados y ser consistentes con las características del tipo de caso correspondiente. Las características las podemos ver con view-type tipoCaso.

(defun put-description (case-name case-description))

(defun put-solution (case-name case-solution))

(defun put-result (case-name case-result)

Los individuos descripción, solución y resultado se pueden crear usando las tres funciones siguientes.

(defun create-description (desc-name tipoDescripción))

(defun create-solution (desc-name tipoDescripción))

(defun create-result (desc-name tipoDescripción))

(defun drop-case (case-name))

Función que elimina un caso.

(defun drop-case-type (case-name case-type))

Función que elimina la pertenencia de un caso a un tipo (sólo será efectivo si la pertenencia es asertada y no inferida)

(defun create-query-type (query-name))

Función que define un tipo de consultas.

3. Módulo de selección y configuración de métodos

Este módulo agrupa las funciones de configuración de métodos y de especificación de requisitos.

(defun create-relevance-criteria ())

Función para definir un criterio de relevancia. Puede definirse en base a alguno de los predefinidos o no.

(defun view-all-relevance-criteria ())

Muestra todos los criterios de relevancia definidos actualmente en el sistema. Subrelaciones de Relevance-Criteria. Inicialmente hay varios predefinidos.

(defun createSimilarityFunction ("Especificación de función de similitud"))

Dada una especificación para una función de similitud, creamos una instancia del concepto primitivo de CBROnto Similarity-Functions con las características dadas.

(defun createSimilarityMeasure (nombre "Especificación de medida de similitud"))

Dada una especificación para una medida de similitud, creamos una instancia del concepto primitivo de CBROnto Similarity-Measure con las características dadas, que incluyen el nombre de la función de la instancia función de similitud que se usará para contenidos, estructura y combinación y la instancia tipo de similitud. Todas estas instancias deben existir previamente (precondición del método).

```
    Ejemplo: (CreateSimilarityMeasure palabra_sim
            :contents iigual :position ideep :combination imedia)
(defun Create_Index_Type (name index_restrictions))
    Creación de un tipo de índice con el nombre y las restricciones dadas.
    <restriction> ::= (<relación> [,<concepto>])+
(defun ViewSimilarityType SM)
    Dada una medida de similitud (nombre de la instancia) muestra sus características de
    tipo de similitud.
(defun ViewContentSimilarityFunction SM)
    Dada una medida de similitud muestra las características de su función de similitud
    por contenidos.
(defun ViewPositionSimilarityFunction SM)
    Dada una medida de similitud muestra las características de su función de similitud
    por posición.
(defun ViewCombinationSimilarityFunction SM)
    Dada una medida de similitud muestra las características de su función de similitud
    de combinación que combina los resultados de posición y contenidos.
(defun ViewAllSimilarityFunctions )
    Muestra las características de todas las instancias de función de similitud.

(defun ViewAllSimilarityMeasures)
    Muestra las características de todas las instancias de medida de similitud,
    incluyendo a qué conceptos están asociadas cada una.
(defun AssociateSimilarityMeasuretoConcept C SM)
    Asocia una medida de similitud a un concepto. Será la medida que se use para
    comparar dos instancias de ese concepto.
(defun ViewConceptSimilarity C )
    Muestra la medida de similitud asociada al concepto C
(defun AssociateIndextoConcept C I)
    Función que asocia un tipo de índice a un concepto
(defun iscycle (iM))
    Función que comprueba si el método de descomposición genera un ciclo entre sus
    subtareas o hace ejecución secuencial de las mismas. Los ciclos se representan por
    clasificación. Un método cicla si es instancia de CYCLE_METHOD.
(defun find-suitable-methods (iT))
    Encuentra metodos cuya competencia sea adecuada para resolver la tarea it.
(defun find-applicable-methods (iT))
    Función para encontrar métodos cuya competencia sea adecuada para resolver la tarea
    it que sean aplicables para el contexto actual (representado por el individuo c).
(defun design-resolve (iT))
    Función que emula la resolucion de tareas para hacer la configuración durante la
    fase de diseño.  es el equivalente a resolve del módulo de resolución de problemas.
(defun tipoCBROnto (iT &key (asserted nil)))
    Devuelve el tipo de la instancia iT dentro de CBROnto. Es decir, el tipo usando solo
    conceptos de CBROnto.
(defun tipoDomain (iT  &key (asserted nil))
   Devuelve el tipo de la instancia iT dentro del dominio. Es decir, el tipo usando solo
conceptos clasificados bajo domain-concept.
(defun forget_task_method (iT iM &key (all t)))
   Funcion para eliminar el vinculo task_method de la tarea iT con el método iM.
```

```
   Si el parametro all vale t se eliminan todos los vínculos task_method de la tarea.
(defun link_task_method (it im))
   Funcion para establecer un enlace task_method entre la tarea it y el método im
   teniendo en cuenta que se deben satisfacen los compromisos ontológicos establecidos.
(defun forget_subtask (iM isT))
   Funcion para eliminar una subtarea de un metodo de descomposicion. Realmente lo que
   ocurre es que la tarea se vincula a través de la relación task_method con el método
   ido_nothing por lo que es cómo si no formara parte del método de descomposición.
   Ejemplo de llamada: (forget_subtask 'iCBR_Method 'iReuse_task)
(defun add_subtask (iM isT))
    Funcion para añadir una subtarea a un metodo de descomposicion. Elimino el vínculo
    task_method con el método ido_nothing si existe y no hace nada e.o.c.
(defun list_subtasks (iM))
   Devuelve la lista de subtareas de un metodo de descomposicion.
   Ejemplo de llamada: (list_subtasks 'iCBR_Method)
(defun new_CBR_cycle ())
   Nuevo ciclo: crea instancias directas de CBR_Task y CBR_Method con los valores por
   defecto adecuados. Devuelve la tarea. El metodo esta accesible a traves del enlace
   task_method. Tambien crea instancias de cada una de las tareas y subtareas.
(defun associate-cycle (iCt T))
    Funcion para asociar un ciclo CBR (iCt instancia de CBR_TASK) con el concepto T que
    debe ser un tipo de caso (subconcepto de CASE), o de consulta (subconcepto de QUERY)
    o de usuario (Subconcepto de USER_TYPES)
(defun free-cycle (iCt))
    Libera la asociación del ciclo iCT.
(defun view-cycles ())
    Muestra todas las instancias de CBR_Task que representan los ciclos, así como el
    concepto al que están asociadas (si existe).
(defun copy-method (m))
    Función que hace una copia de un método (y de sus subtareas y opciones
    recursivamente) Esto hace falta por ejemplo para poder usar métodos sueltos (fuera
    de la secuencia) en las estrategias de adaptación y reparación.
(defun which-input-requirements (iM)))
   Dado un metodo devuelve una lista con los requisitos de conocimiento qeu debe
   satisfacer. Devuelve una lista de relaciones que hay que rellenar convenientemente
   usando el metodo put-input-requirements
(check-input-requirements (iM))
    Genera un diálogo que muestra los requisitos de entrada establecidos y permite
    modificarlos.
(defun put-design-requirements (iM lista))
    Establece los requisitos de diseño en el método iM. Que son parte de los requisitos
    de entrada del método.
(defun put-parameter-requirements (iM lista))
   Establece los requisitos paramétricos en el método iM. También son parte de los
   requisitos de entrada del método.
(defun free_input-requirements (iM))
   Elimina todos los requisitos de entrada que estuvieran configurados en el método. Lo
   deja inicializado.
(defun competencia (iM))
   devuelve cualquier instancia que pueda servir como competencia del metodo.
(defun create-adaptation-strategy (name &key (function-name) &key (method-name)))))
```

Función que crea una estrategia de reparación de nombre name (si no existe) y la modifica en otro caso. El resultado es una instancia de Adaptation-Strategy que hace referencia a su especificación operacional a través de una función, un método configurado o ambos, que deben estar creados previamente.

(defun delete-adaptation-strategy (name))

Función que elimina una estrategia de adaptación.

(defun create-repair-strategy (name &key (function-name) &key (method-name)))))

Función que crea una estrategia de reparación de nombre name (si no existe) y la modifica en otro caso. El resultado es una instancia de Repair-STrategy que hace referencia a su especificación operacional a través de una función, un método configurado o ambos, que deben estar creados previamente.

(defun delete-repair-strategy (name))

Función que elimina una estrategia de reparación.

(defun create-problem-type ())

Función que crea un tipo de problema (para la adaptación). Es un subconcepto de Problem-Type

(defun create-fail-type ())

Función que crea un tipo de fallo (para evaluación dentro de la revision). Instancia de Fail_Type.

(defun associate-repair-strategy (ft rs))

Función que asocia una estrategia de reparación con un tipo de fallo. La estrategia de reparación st es una instancia del concepto de CBROnto repair_strategy y se une con el tipo de fallo ft a través de la relación de CBROnto has_repair_strategy.

(defun view-adaptation-strategy (st))

Función para ver las características de una estrategia de adaptación. Muestra las características de cada una de sus transformaciones haciendo llamadas a la función view-Transformation.

(defun view-Transformation (t))

Función que muestra (interfaz textual) o devuelve (interfaz gráfica) las características de la transformación (instancia de Transformation) dada.

(defun create-simple-search-strategy (name &key (begin-with)))

Función para definir una estrategia de búsqueda de sustitutos en base a las operaciones de acceso a memoria (instancia de Simple-Search-Strategy). Crea una instancia con el punto de origen. Hay que añadir las primitivas de acceso a memoria con la función siguietne.

(defun make-search-step (name primitive-step))

Función para crear un paso de búsqueda (instancia de Search-Step)

(defun add-search-step (SS &key (search-step)))

Función que añade un paso de búsqueda (instancia de Search-Step) a la estrategia de búsqueda SS (debe ser instancia de Simple-Search-Strategy).

(defun create-method-search-strategy (name weigth search-substitutes-method))

Función que crea una estrategia de busqueda de sustitutos basada en un método de CBROnto. Debe ser un método de búsqueda de sustitutos adecuado, es decir, una isntancia de Search-Substitutes-Method

(defun create_add_transformation (applicability add-search-strategy weight))

Función que crea una transformación de inserción (instancia de Add_Transformation)

applicability es una lista de la forma:

({relation-name [<concept-name>]}+ | <concept-name> | <individual-name>)

y add-search-strategy es una instancia de Simple-Search-Strategy (que se crea usando la función create-simple-search-strategy descrita)

(defun create_delete_transformation (applicability weight))

Función que crea una transformación de borrado (instancia de Delete_Transformation)

(defun create_substitute_transformation (applicability search-strategy weight))

Función que crea una transformación de sustitución.

search-strategy es una instancia de Search-Strategy, tanto una instancia de Simple-Search-Strategy como de Method-Search-Strategy (los dos subconceptos de Search-Strategy que forman una partición exhaustiva).

4. Módulo de resolución de problemas

(defun resolve (iT))

Resolutor de tareas CBR de la interfaz textual en Lisp.

(el resolutor de tareas correspondiente a la interfaz gráfica está en la parte Java)

(defun resolve-subtask-list (lista-Tareas))

Resuelve una lista de subtareas pasando a cada una el resultado de la tarea anterior

(defun retrieve-task-method (iT))

Función que comprueba si la tarea iT tiene enlaces task_method es decir algun o algunos metodos elegido para resolverla. Recupera los métodos asociados a una tarea para su resolución

(defun aplicar-res (iM res))

Aplica un metodo de resolucion. Res es la salida del método que resuelve la tarea anterior de la secuencia. Será nil si es la primera.

(defun aplicar-desc (iM))

Se encarga de aplicar un metodo de descomposicion, y como resultado devuelve la secuencia de tareas a resolver

(defun resolution (iM))

Devuelve t si iM es un metodo de resolución y nil si es un metodo de descomposición.

(defun elige (mensaje lista))

Muestra una lista (con título mensaje) para que el usuario elija una de las opciones.

(defun create-query ())

Función que define una consulta de usuario. Se crea una instancia de Query y del tipo de caso.

(defun show-current-context ())

Función que muestra el contexto actual. Muestra la información asertada en el individuo current-context que depende de las características de la base de conocimiento actual.

(defun update-current-context ())

Función que actualiza la información sobre el contexto actual. Obtiene informacion del conocimeinto representado y hace los asertos correspondientes en el individuo current-context

APÉNDICE E: GENERACIÓN DE POESÍA

1. Base de conocimiento del dominio de la poesía en castellano

```
(defconcept Prosa_Concept
    :is-primitive (:and domain-concept
            (:at-least 1 tiene-linea)
            (:all tiene-linea LineaTexto_Concept)))

(defconcept LineaTexto_Concept
    :is-primitive (:and domain-concept
                        (:at-least 1 tiene-palabra)
                        (:all tiene-palabra Aparicion-Palabra)))

(defconcept Poema_Concept
    :is-primitive (:and domain-concept
                        (:at-least 1 tiene-estrofa)
                        (:all tiene-estrofa Estrofa_Concept)))

(defconcept Estrofa_Concept
    :is-primitive (:and domain-concept
                        (:at-least 1 tiene-verso)))

(defconcept SinEstructura :is-primitive Estrofa_Concept)

;; En vez de ver el soneto como una estrofa de 14 versoslo vemos como una estrofa
;; que a su vez tiene otras estrofas, que son 2 cuartetos y 2 tercetos.
;; Aparte cada terceto y cada cuarteto son poemas independientes.
;; Esto es un ejemplo de casos como parte de otros casos.

(defconcept Soneto_Concept
    :is (:and Poema_Concept
            (:exactly 1 primer-cuarteto) (:exactly 1 segundo-cuarteto)
            (:exactly 1 primer-terceto) (:exactly 1 segundo-terceto)
            (:the primer-terceto Terceto_Uno_Tres)
            (:the segundo-terceto Terceto_Uno_Tres)))

(defconcept Soneto_Tipo1
    :is (:and Soneto_Concept
            (misma-rima primer-cuarteto segundo-cuarteto)
            (misma-rima primer-terceto segundo-terceto)))

(defconcept Soneto_Tipo2
    :is (:or (:and Soneto_Concept
                    (misma-rima primer-cuarteto segundo-cuarteto)
                    (rima-inversa primer-terceto segundo-terceto))
            (:and Poema_Concept
                (:exactly 1 primer-cuarteto)
                (:exactly 1 segundo-cuarteto)
                (:some tiene-estrofa Terceto_Encadenado))))
;; las relaciones primer-cuarteto, segundo-cuarteto, primer-terceto y segundo-terceto
son subrelaciones de tiene-estrofa, lo que permite manejarlas de forma conjunta o
separada según se necesite.
```

```
;; Los romances son estrofas que tienen la longitud que tu quieras.
;; cualquier numero de par de versos tal que riman los versos pares con rima asonante.
;; Los versos son octosílabos en vez de endecasilabos como los anteriores.
(defconcept Romance_Concept
:is-primitive (:or  (:and Estrofa_Concept
                          (:all tiene-verso Octosilabo)
                          (:exactly 1 primer-verso)(:exactly 1 segundo-verso)
                          (:exactly 1 tercer-verso)(:exactly 1 cuarto-verso))
                    (:and Estrofa_Concept (:all tiene-estrofa Romance_Concept)))

(defconcept Cuarteto_Concept
    :is (:and Estrofa_Concept
            (:all tiene-verso Endecasilabo)
            (:exactly 1 primer-verso)(:exactly 1 segundo-verso)
            (:exactly 1 tercer-verso)(:exactly 1 cuarto-verso)
            (:relates riman primer-verso cuarto-verso)
            (:relates riman segundo-verso tercer-verso))
 :implies (:and (:all tiene-verso Verso_Rimado)))
;; las relaciones primer-verso, segundo-verso, tercer-verso y cuarto-verso son
;; subrelaciones de tiene-verso, lo que permite manejarlas de forma conjunta o separada
;; según se necesite.

(defconcept Terceto_Concept
    :is (:and Estrofa_Concept
            (:all tiene-verso Endecasilabo)
            (:exactly 1 primer-verso)
            (:exactly 1 segundo-verso)
            (:exactly 1 tercer-verso)))

;; TercetoUnoTres reconoce a los tercetos en los que rima el primer verso con
;; el tercero.
(defconcept TercetoUnoTres
    :is (:and Terceto_Concept
            (:relates riman primer-verso tercer-verso))
    :implies (:and  (:the primer-verso Verso_Rimado)
                    (:the tercer-verso Verso_Rimado)
                    (:the segundo-verso Verso_No_Rimado)))

(defconcept Terceto_Encadenado
    :is (:and Poema_Concept
            (:exactly 1 primer-terceto)
            (:exactly 1 segundo-terceto)
            (rima-inversa primer-terceto segundo-terceto)))

(defconcept Categoria_Concept :is-primitive domain-concept)

(defconcept Palabra_Concept
    :is-primitive (:and domain-concept
                      (:the textoPalabra String) ;; nombre escrito de la letra.
                      (:all categoria-sintactica Categoria_Concept)
                      (:the numero-silabas-palabra Number)
                      (:the acento Number)
                      (:the rima-palabra String)
                      (:the empVocal Number) ;;boolean
                      (:the terVocal Number) ;;boolean
                      ))
```

```
(defconcept Aparicion-Palabra
    :is-primitive (:and domain-concept
                        (:exactly 1 de-palabra)
                        (:all de-palabra Palabra_Concept)
                        (:at-most 1 anterior-a-palabra)))

(defconcept Aparicion-Palabra-Rimada
:is (:and Palabra_Final_Linea
     (:satisfies (?x) (:and (Palabra_Final_Linea ?x)
              (:for-some (?y)
                     (:and (Verso_Rimado ?y)
                       (ultima-palabra ?y ?x)))))))

(defconcept Aparicion-Palabra-No-Rimada
:is (:and  Palabra_Final_Linea
       (:satisfies (?x) (:and (Palabra_Final_Linea ?x)
                   (:fail (Aparicion-Palabra-Rimada ?x))))))

(defconcept Aparicion_Termina_Vocal
    :is (:and Aparicion-Palabra
            (:the de-palabra Termina_Vocal)))

(defconcept Aparicion_No_Termina_Vocal
    :is (:and Aparicion-Palabra
            (:the de-palabra No_Termina_Vocal)))

(defconcept Aparicion_Empieza_Vocal
    :is (:and Aparicion-Palabra
            (:the de-palabra Empieza_Vocal)))

(defconcept Aparicion_No_Empieza_Vocal
    :is (:and Aparicion-Palabra
            (:the de-palabra No_Empieza_Vocal)))

;; La primera palabra de una sinalefa termina en vocal y la siguiente empieza por vocal.
(defconcept Aparicion-Palabra-Principio-Sinalefa
    :is (:and   Aparicion-Palabra
                Aparicion_Termina_Vocal
            (:the anterior-a-palabra Aparicion_Empieza_Vocal))
    :implies (:the anterior-a-palabra  Aparicion-Palabra-Fin-Sinalefa))
(defconcept Aparicion-Palabra-Fin-Sinalefa)

(defconcept Verso_Concept
    :is-primitive   (:and LineaTexto_Concept
              (:the numero-silabas-verso Number)
              ;; se puede obtener de sus palabras
              (:the rima-verso String)  ;; se puede obtener de la ultima palabra
              ;; representar coincide con la rima de su ultima palabra.
              ;; tiene-palabra lo hereda de LineaTexto.
              (:all patron-acentos PatronAcentos)
              (:at-least 1 patron-acentos)
              (:at-most 1 encabalgado-con)))

(defconcept Verso_Rimado :is-primitive (:and Verso_Concept))
(defconcept Verso_No_Rimado :is-primitive (:and Verso_Concept))

(defconcept Heptasilabo :is (:and Verso_Concept(:the numero-silabas-verso 7)))
```

```
(defconcept Octosilabo :is (:and Verso_Concept (:the numero-silabas-verso 8)))
(defconcept Eneasilabo :is (:and Verso_Concept (:the numero-silabas-verso 9)))
(defconcept Decasilabo :is (:and Verso_Concept (:the numero-silabas-verso 10)))
(defconcept Endecasilabo :is (:and Verso_Concept (:the numero-silabas-verso 11)))
(defconcept Dodecasilabo :is (:and Verso_Concept (:the numero-silabas-verso 12)))

(defconcept PatronAcentos :is-primitive domain-concept)

(defconcept Enfatico :is (:and Endecasilabo
                  (:some patron-acentos Patron_uno_seis_diez)))
(defconcept Heroico :is (:and Endecasilabo
                 (:some patron-acentos Patron_dos_seis_diez)))
(defconcept Melodico :is (:and Endecasilabo
                  (:some patron-acentos Patron_tres_seis_diez)))
(defconcept Safico :is (:and Endecasilabo
                (:or (:some patron-acentos Patron_cuatro_seis_diez)
                 (:some patron-acentos Patron_cuatro_ocho_diez))))

;; Pablo comento algo de representar los patrones como numeros decimales que
corresponden a la
;; representacion binaria del patron. Si es mejor cambiarlo.

(defconcept Patron_uno_seis_diez :is-primitive PatronAcentos)
(defconcept Patron_dos_seis_diez :is-primitive PatronAcentos)
(defconcept Patron_tres_seis_diez :is-primitive PatronAcentos)
(defconcept Patron_cuatro_seis_diez :is-primitive PatronAcentos)
(defconcept Patron_cuatro_ocho_diez :is-primitive PatronAcentos)

(tell (:about uno_seis_diez Patron_uno_seis_diez))
(tell (:about dos_seis_diez Patron_Dos_seis_diez))
(tell (:about tres_seis_diez Patron_Tres_seis_diez))
(tell (:about cuatro_seis_diez Patron_Cuatro_seis_diez))
(tell (:about cuatro_ocho_diez Patron_Cuatro_ocho_diez))

(defconcept Termina_Vocal
    :is (:and Palabra_Concept (:filled-by terVocal 1)))
(defconcept No_Termina_Vocal
    :is (:and Palabra_Concept (:filled-by terVocal 0)))
(defconcept Empieza_Vocal
    :is (:and Palabra_Concept (:filled-by empVocal 1)))
(defconcept No_Empieza_Vocal
    :is (:and Palabra_Concept (:filled-by empVocal 0)))
(defconcept Una_Silaba
    :is (:and Palabra_Concept (:filled-by numero-silabas-palabra 1)))

(defconcept Dos_Silabas
    :is  (:and Palabra_Concept (:filled-by numero-silabas-palabra 2)))
(defconcept Tres_Silabas
    :is (:and Palabra_Concept (:filled-by numero-silabas-palabra 3)))
(defconcept Cuatro_Silabas
    :is (:and Palabra_Concept (:filled-by numero-silabas-palabra 4)))
(defconcept Cinco_Silabas
    :is (:and Palabra_Concept (:filled-by numero-silabas-palabra 5)))
(defconcept Seis_Silabas
    :is (:and Palabra_Concept (:filled-by numero-silabas-palabra 6)))
(defconcept Palabra_Final_Linea
    :is (:and Aparicion-Palabra (:exactly 0 anterior-a-palabra)))
```

```
(defconcept Palabra_No_Final_Linea
    :is (:and Aparicion-Palabra  (:at-least 1 anterior-a-palabra)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;; relaciones ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defrelation tiene-linea
    :is-primitive (:and domain-relation has_part)
    :range LineaTexto_Concept)

(defrelation tiene-palabra
    :is-primitive (:and domain-relation has_part)
    :domain LineaTexto_Concept
    :characteristics (:closed-world))

 (defrelation primera-palabra
    :is-primitive (:and domain-relation has_part)
    :domain LineaTexto_Concept)

(defrelation ultima-palabra
    :domain LineaTexto_Concept
    :range Aparicion-Palabra
    :is (:satisfies (?x ?y)
            (:and  (tiene-palabra ?x ?y)
                (:fail
                 (:for-some ?z
                    (anterior-a-palabra ?y ?z))))))
;; Ejemplo de llamada: (retrieve ?x (ultima-palabra VERSO1_ESTROFA1_POSGDLVC2 ?x))
;; (|I|VENA232)

(defrelation tiene-estrofa
    :is-primitive (:and domain-relation has_part)
    :domain Poema_Concept)
(defrelation tiene-verso
    :is-primitive domain-relation
    :range Verso_Concept
    :characteristics (:closed-world))

(defrelation primer-verso
    :is-primitive tiene-verso
    :characteristics (:closed-world))

(defrelation segundo-verso
    is-primitive tiene-verso
    :characteristics (:closed-world))

(defrelation tercer-verso
    :is-primitive tiene-verso
    :characteristics (:closed-world))

(defrelation cuarto-verso
    :is-primitive tiene-verso
    :characteristics (:closed-world))

(defrelation primer-cuarteto
    :is-primitive tiene-estrofa
    :characteristics (:closed-world))
```

```
(defrelation segundo-cuarteto
    :is-primitive tiene-estrofa
    :characteristics (:closed-world))

(defrelation primer-terceto
    :is-primitive tiene-estrofa
    :characteristics (:closed-world))

(defrelation segundo-terceto
    :is-primitive tiene-estrofa
    :characteristics (:closed-world))

(defrelation anterior-a-palabra
    :is-primitive (:and domain-relation before)
    :domain Aparicion-Palabra
    :range Aparicion-Palabra
    :characteristics (:closed-world))

(defrelation de-palabra
    :is-primitive domain-relation
    :domain Aparicion-Palabra
    :range Palabra_Concept)

(defrelation empVocal
    :is-primitive domain-relation
    :domain Palabra_Concept :range Number
     :characteristics (:closed-world))

(defrelation terVocal
    :is-primitive domain-relation
    :domain Palabra_Concept :range Number
     :characteristics (:closed-world))

(defrelation numero-silabas-palabra
    :is-primitive domain-relation
    :domain Palabra_Concept :range Number
     :characteristics (:closed-world))

(defrelation acento
    :is-primitive domain-relation
    :domain Palabra_Concept :range Number)

(defrelation categoria-sintactica
    :is-primitive domain-relation
    :domain Palabra_Concept
    :range Categoria_Concept
     :characteristics (:closed-world))

(defrelation textoPalabra
    :is-primitive domain-relation
    :domain Palabra_Concept :range String)

(defrelation rima-palabra
    :is-primitive domain-relation
    :domain Palabra_Concept :range String)
(defrelation rima-verso
    :is-primitive domain-relation
```

```
    :domain Verso_Concept
    :range String
     :characteristics (:closed-world))

(defrelation riman
    :is (:satisfies (?x ?y)
            (:and (Verso_Concept ?x)
              (Verso_Concept ?y)
              (:for-some ?z (:and (rima-verso ?x ?z) (rima-verso ?y ?z)))
              )))
;; La instrucción siguiente recupera todos los versos que riman:
;; (retrieve (?x ?y) (riman ?x ?y))

(defrelation patron-acentos
    :is-primitive domain-relation
    :domain Verso_Concept
    :range PatronAcentos
     :characteristics (:closed-world))

(defrelation numero-silabas-verso
    :is-primitive domain-relation
    :domain Verso_Concept
    :range Number
     :characteristics (:closed-world))

(defrelation encabalgado-con
    :is-primitive (:and domain-relation before depends_on)
    :range Verso_Concept)

(finalize-definitions)
```

2. Categorías sintácticas

```
(defconcept CAT_PE :is-primitive Categoria_Concept
:annotations ((documentation " Foreign word  ")))
(tell (:about PE CAT_PE (documentation " Foreign word  ")))

(defconcept CAT_PNC :is-primitive Categoria_Concept
:annotations ((documentation " Unclassified word ")))
(tell (:about PNC CAT_PNC (documentation " Unclassified word ")))

(defconcept CAT_FO :is-primitive Categoria_Concept
:annotations ((documentation " Formula ")))
(tell (:about FO CAT_FO (documentation " Formula ")))

(defconcept CAT_CODE :is-primitive Categoria_Concept
:annotations ((documentation " Alphanumeric code ")))
(tell (:about CODE CAT_CODE (documentation " Alphanumeric code ")))

(defconcept CAT_ACRNM :is-primitive Categoria_Concept
:annotations ((documentation " Acronym (ADN) ")))
(tell (:about ACRNM CAT_ACRNM (documentation " Acronym (ADN) ")))

(defconcept CAT_NEG :is-primitive Categoria_Concept
:annotations ((documentation " Negation ")))
(tell (:about NEG CAT_NEG (documentation " Negation ")))
```

```
(defconcept CAT_ITJN :is-primitive Categoria_Concept
:annotations ((documentation " Interjection (oh, ja) ")))
(tell (:about ITJN CAT_ITJN (documentation " Interjection (oh, ja) ")))

(defconcept CAT_SE :is-primitive Categoria_Concept
:annotations ((documentation " Se (as particle) ")))
(tell (:about SE CAT_SE (documentation " Se (as particle) ")))

(defconcept CAT_CONTART :is-primitive Categoria_Concept
:annotations ((documentation " Contracciones con articulo ")))

(defconcept CAT_PAL :is-primitive CAT_CONTART
:annotations ((documentation " Portmanteau word formed by a and el  ")))
(tell (:about PAL CAT_PAL (documentation " Portmanteau word formed by a and el  ")))

(defconcept CAT_PDEL :is-primitive CAT_CONTART
:annotations ((documentation " Portmanteau word formed by de and el  ")))
(tell (:about PDEL CAT_PDEL (documentation " Portmanteau word formed by de and el  ")))

(defconcept CAT_PREPN :is-primitive CAT_PREP :annotations ((documentation " Negative
preposition (sin) ")))
(tell (:about PREPN CAT_PREPN (documentation " Negative preposition (sin) ")))

(defconcept CAT_CONJ :is-primitive Categoria_Concept :annotations ((documentation "
Conjunción ")))

(defconcept CAT_CC :is-primitive CAT_CONJ :annotations ((documentation " Coordinating
conjunction (y, o)  ")))
(tell (:about CC CAT_CC (documentation " Coordinating conjunction (y, o)  ")))

(defconcept CAT_CCNEG :is-primitive CAT_CONJ :annotations ((documentation " Negative
coordinating conjunction (ni)pero)  ")))
(tell (:about CCNEG CAT_CCNEG (documentation " Negative coordinating conjunction
(ni)pero)  ")))

(defconcept CAT_CSUBF :is-primitive CAT_CONJ :annotations ((documentation "
Subordinating conjunction that introduces finite clauses (apenas)  ")))
(tell (:about CSUBF CAT_CSUBF (documentation " Subordinating conjunction that introduces
finite clauses (apenas)  ")))

(defconcept CAT_CSUBI :is-primitive CAT_CONJ :annotations ((documentation "
Subordinating conjunction that introduces infinite clauses (al) )  ")))
(tell (:about CSUBI CAT_CSUBI (documentation " Subordinating conjunction that introduces
infinite clauses (al) )  ")))

(defconcept CAT_CSUBX :is-primitive CAT_CONJ :annotations ((documentation "
Subordinating conjunction underspecified for subord-type (aunque)  ")))
(tell (:about CSUBX CAT_CSUBX (documentation " Subordinating conjunction underspecified
for subord-type (aunque)  ")))

(defconcept CAT_ART :is-primitive Categoria_Concept :annotations ((documentation "
Artículos ")))

(defconcept CAT_ARCAFS :is-primitive CAT_ART :annotations ((documentation " Feminine
singular indefinite article and cardinal capable of pronominal function (una)  ")))
(tell (:about ARCAFS CAT_ARCAFS (documentation " Feminine singular indefinite article
and cardinal capable of pronominal function (una)  ")))
```

```
(defconcept CAT_ARCAMS :is-primitive CAT_ART :annotations ((documentation " Masculine
singular indefinite article and non pronominal cardinal (un)  function (una)  ")))
(tell (:about ARCAMS CAT_ARCAMS (documentation " Masculine singular indefinite article
and non pronominal cardinal (un)  function (una)  ")))

(defconcept CAT_ARQUFP :is-primitive CAT_ART :annotations ((documentation " Feminine
plural indefinite article and quantifier capable of pronominal function (unas)  ")))
(tell (:about ARQUFP CAT_ARQUFP (documentation " Feminine plural indefinite article and
quantifier capable of pronominal function (unas)  ")))

(defconcept CAT_ARQUMP :is-primitive CAT_ART :annotations ((documentation " Masculine
plural indefinite article and quantifier capable of pronominal function (unos)  ")))
(tell (:about ARQUMP CAT_ARQUMP (documentation " Masculine plural indefinite article and
quantifier capable of pronominal function (unos)  ")))

(defconcept CAT_ARTDFP :is-primitive CAT_ART :annotations ((documentation " Feminine
plural definite article (las)  quantifier capable of pronominal function (unos)  ")))
(tell (:about ARTDFP CAT_ARTDFP (documentation " Feminine plural definite article (las)
quantifier capable of pronominal function (unos)  ")))

(defconcept CAT_ARTDFS :is-primitive CAT_ART :annotations ((documentation " Feminine
singular definite article (la) quantifier capable of pronominal function (unos)  ")))
(tell (:about ARTDFS CAT_ARTDFS (documentation " Feminine singular definite article (la)
quantifier capable of pronominal function (unos)  ")))

(defconcept CAT_ARTDMP :is-primitive CAT_ART :annotations ((documentation " Masculine
plural definite article (los) quantifier capable of pronominal function (unos)  ")))
(tell (:about ARTDMP CAT_ARTDMP (documentation " Masculine plural definite article (los)
quantifier capable of pronominal function (unos)  ")))

(defconcept CAT_ARTDMS :is-primitive CAT_ART :annotations ((documentation " Masculine
singular definite article (el) uantifier capable of pronominal function (unos)  ")))
(tell (:about ARTDMS CAT_ARTDMS (documentation " Masculine singular definite article
(el) uantifier capable of pronominal function (unos)  ")))

(defconcept CAT_ARTDNS :is-primitive CAT_ART :annotations ((documentation " Neuter
singular definite article (lo)el) uantifier capable of pronominal function (unos)  ")))
(tell (:about ARTDNS CAT_ARTDNS (documentation " Neuter singular definite article
(lo)el) uantifier capable of pronominal function (unos)  ")))

(defconcept CAT_NOM :is-primitive Categoria_Concept :annotations ((documentation "
Nombres ")))
(defconcept CAT_NC :is-primitive CAT_NOM :annotations ((documentation " Nombres comunes
")))
(defconcept CAT_NCFP :is-primitive CAT_NC :annotations ((documentation " Feminine plural
common noun (mesas, manos)  ")))
(tell (:about NCFP CAT_NCFP (documentation " Feminine plural common noun (mesas, manos)
")))
(defconcept CAT_NCFS :is-primitive CAT_NC :annotations ((documentation " Feminine
singular common noun (mesa, mano)  ")))
(tell (:about NCFS CAT_NCFS
               (documentation " Feminine singular common noun (mesa, mano)  ")))
(defconcept CAT_NCMP :is-primitive CAT_NC :annotations ((documentation " Masculine
plural common noun (libros, ordenadores)  ")))
(tell (:about NCMP CAT_NCMP (documentation " Masculine plural common noun (libros,
ordenadores)  ")))
```

```
(defconcept CAT_NCMS :is-primitive CAT_NC :annotations ((documentation " Masculine
singular common noun (libro, ordenador))  ")))
(tell (:about NCMS CAT_NCMS (documentation " Masculine singular common noun (libro,
ordenador))  ")))

(defconcept CAT_UNI :is-primitive CAT_NOM :annotations ((documentation " Unidades ")))

(defconcept CAT_UMFX :is-primitive CAT_UNI :annotations ((documentation " Feminine unit
of measurement, neutral for number (pta.)  ")))
(tell (:about UMFX CAT_UMFX (documentation " Feminine unit of measurement, neutral for
number (pta.)  ")))

(defconcept CAT_UMMX :is-primitive CAT_UNI :annotations ((documentation " Masculine unit
of measurement, neutral for number (cm.)  ")))
(tell (:about UMMX CAT_UMMX (documentation " Masculine unit of measurement, neutral for
number (cm.)  ")))

(defconcept CAT_NMEAFP :is-primitive CAT_UNI :annotations ((documentation " Feminine
plural unit of measure noun (hect'areas, micras)  ")))
(tell (:about NMEAFP CAT_NMEAFP (documentation " Feminine plural unit of measure noun
(hect'areas, micras)  ")))

(defconcept CAT_NMEAFS :is-primitive CAT_UNI :annotations ((documentation " Feminine
singular unit of measure noun (hect'area, micra)  ")))
(tell (:about NMEAFS CAT_NMEAFS (documentation " Feminine singular unit of measure noun
(hect'area, micra)  ")))

(defconcept CAT_NMEAMP :is-primitive CAT_UNI :annotations ((documentation " Masculine
plural unit of measure noun (metros, litros) a)  ")))
(tell (:about NMEAMP CAT_NMEAMP (documentation " Masculine plural unit of measure noun
(metros, litros) a)  ")))

(defconcept CAT_NMEAMS :is-primitive CAT_UNI :annotations ((documentation " Masculine
singular unit of measure noun (metro, litro) a)  ")))
(tell (:about NMEAMS CAT_NMEAMS (documentation " Masculine singular unit of measure noun
(metro, litro) a)  ")))

 (defconcept CAT_NPAFS :is-primitive CAT_NANT :annotations ((documentation " Feminine
singular proper anthroponymous noun (Mar'ia)  ")))
(tell (:about NPAFS CAT_NPAFS (documentation " Feminine singular proper anthroponymous
noun (Mar'ia)  ")))

(defconcept CAT_NPAMP :is-primitive CAT_NANT :annotations ((documentation " Masculine
plural proper anthroponymous noun (Juanes)   ")))
(tell (:about NPAMP CAT_NPAMP (documentation " Masculine plural proper anthroponymous
noun (Juanes)   ")))

(defconcept CAT_NPAMS :is-primitive CAT_NANT :annotations ((documentation " Masculine
singular proper anthroponymous noun (Juan)   ")))
(tell (:about NPAMS CAT_NPAMS (documentation " Masculine singular proper anthroponymous
noun (Juan)   ")))

(defconcept CAT_NPAXX :is-primitive CAT_NANT :annotations ((documentation " Proper
anthroponymous noun neutral for gender and number (Rodr'iguez, Sanch'is)  ")))
(tell (:about NPAXX CAT_NPAXX (documentation " Proper anthroponymous noun neutral for
gender and number (Rodr'iguez, Sanch'is)  ")))
```

```
(defconcept CAT_NPOS :is-primitive CAT_NOTROS :annotations ((documentation " Singular
proper collective nouns (Iberia) n proper nouns)  ")))
(tell (:about NPOS CAT_NPOS (documentation " Singular proper collective nouns (Iberia) n
proper nouns)  ")))

(defconcept CAT_NTOP :is-primitive CAT_NOM :annotations ((documentation " Nombres
topónimos ")))

(defconcept CAT_NPTOP :is-primitive CAT_NTOP :annotations ((documentation " Plural
proper toponym or collective noun (Coreas)  ")))
(tell (:about NPTOP CAT_NPTOP (documentation " Plural proper toponym or collective noun
(Coreas)  ")))

(defconcept CAT_NPTOS :is-primitive CAT_NTOP :annotations ((documentation " Singular
proper toponym or collective noun (Madrid)  ")))
(tell (:about NPTOS CAT_NPTOS (documentation " Singular proper toponym or collective
noun (Madrid)  ")))

(defconcept CAT_NPTP :is-primitive CAT_NTOP :annotations ((documentation " Plural proper
toponym noun (Pirineos) noun (Madrid)  ")))
(tell (:about NPTP CAT_NPTP (documentation " Plural proper toponym noun (Pirineos) noun
(Madrid)  ")))

(defconcept CAT_NPTS :is-primitive CAT_NTOP :annotations ((documentation " Singular
proper toponym noun (Guadalquivir) Madrid)  ")))
(tell (:about NPTS CAT_NPTS (documentation " Singular proper toponym noun (Guadalquivir)
Madrid)  ")))

(defconcept CAT_NPT :is-primitive CAT_NOM :annotations ((documentation " Nombres
periodos de tiempo ")))
(defconcept CAT_NWEE :is-primitive CAT_NPT :annotations ((documentation " Weekday nouns
(singular and plural) (s'abado(s)) ")))

(tell (:about NWEE CAT_NWEE (documentation " Weekday nouns (singular and plural)
(s'abado(s)) ")))

(defconcept CAT_NMON :is-primitive CAT_NPT :annotations ((documentation " Month nouns
(singular and plural) (diciembre(s))  ")))
(tell (:about NMON CAT_NMON (documentation " Month nouns (singular and plural)
(diciembre(s))  ")))

(defconcept CAT_NLALF :is-primitive CAT_NOM :annotations ((documentation " Nombres
letras alfabeto ")))

(defconcept CAT_ALFP :is-primitive CAT_NLALF :annotations ((documentation " Plural
letter of the alphabet (As/Aes, bes)  ")))
(tell (:about ALFP CAT_ALFP (documentation " Plural letter of the alphabet (As/Aes, bes)
")))

(defconcept CAT_ALFS :is-primitive CAT_NLALF :annotations ((documentation " Singular
letter of the alphabet (A, b) bes)  ")))
(tell (:about ALFS CAT_ALFS (documentation " Singular letter of the alphabet (A, b) bes)
")))

(defconcept CAT_ADJ :is-primitive Categoria_Concept :annotations ((documentation "
Adjetivos ")))
```

```
(defconcept CAT_ADJCOMP :is-primitive CAT_ADJ :annotations ((documentation " Adejtivos
comparativos ")))

(defconcept CAT_ADJCP :is-primitive CAT_ADJCOMP :annotations ((documentation " Plural
general comparative adjective (mayores, menores)  ")))
(tell (:about ADJCP CAT_ADJCP (documentation " Plural general comparative adjective
(mayores, menores)  ")))

(defconcept CAT_ADJCS :is-primitive CAT_ADJCOMP :annotations ((documentation " Singular
general comparative adjective (mayor, menor) )  ")))
(tell (:about ADJCS CAT_ADJCS (documentation " Singular general comparative adjective
(mayor, menor) )  ")))

(defconcept CAT_ADJPOS :is-primitive CAT_ADJ :annotations ((documentation " Adjetivos
positivos ")))

(defconcept CAT_ADJGFP :is-primitive CAT_ADJPOS :annotations ((documentation " Feminine
plural general positive adjective  ")))
(tell (:about ADJGFP CAT_ADJGFP (documentation " Feminine plural general positive
adjective  ")))

(defconcept CAT_ADJGFS :is-primitive CAT_ADJPOS :annotations ((documentation " Feminine
singular general positive adjective  ")))
(tell (:about ADJGFS CAT_ADJGFS (documentation " Feminine singular general positive
adjective  ")))

(defconcept CAT_ADJGMP :is-primitive CAT_ADJPOS :annotations ((documentation " Masculine
plural general positive adjective   ")))
(tell (:about ADJGMP CAT_ADJGMP (documentation " Masculine plural general positive
adjective   ")))

(defconcept CAT_ADJGMS :is-primitive CAT_ADJPOS :annotations ((documentation " Masculine
singular general positive adjective  ")))
(tell (:about ADJGMS CAT_ADJGMS (documentation " Masculine singular general positive
adjective  ")))

(defconcept CAT_ADJSUP :is-primitive CAT_ADJ :annotations ((documentation " Adjetivos
superlativos ")))

(defconcept CAT_ADJSFP :is-primitive CAT_ADJSUP :annotations ((documentation " Feminine
plural general superlative adjective (m'aximas, m'inimas)  ")))
(tell (:about ADJSFP CAT_ADJSFP (documentation " Feminine plural general superlative
adjective (m'aximas, m'inimas)  ")))

(defconcept CAT_ADJSFS :is-primitive CAT_ADJSUP :annotations ((documentation " Feminine
singular general superlative adjective (m'axima, m'inima)  ")))
(tell (:about ADJSFS CAT_ADJSFS (documentation " Feminine singular general superlative
adjective (m'axima, m'inima)  ")))

(defconcept CAT_ADJSMP :is-primitive CAT_ADJSUP :annotations ((documentation " Masculine
plural general superlative adjective (m'aximos, m'inimos)  ")))
(tell (:about ADJSMP CAT_ADJSMP (documentation " Masculine plural general superlative
adjective (m'aximos, m'inimos)  ")))

(defconcept CAT_ADJSMS :is-primitive CAT_ADJSUP :annotations ((documentation " Masculine
singular general superlative adjective (m'aximo, m'inimo, grand'isimo) ")))
```

```
(tell (:about ADJSMS CAT_ADJSMS (documentation " Masculine singular general superlative
adjective (m'aximo, m'inimo, grand'isimo) ")))

.......Existen muchas más categorías que no incluimos por razones de espacio
```

3. Ejemplo de representación de un caso

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; La representación siguiente se genera automáticamente a partir de
;; una descripción textual del poema.

(tell (:about c1 tipoPoesia (has-description prc1) (has-solution poc1)))
(tell (:about v1e1 Verso (numero-silabas-verso 11) (rima-verso "ida")
                        (tiene-palabra en1) (tiene-palabra mitad1)
                        (tiene-palabra del1) (tiene-palabra camino1)
                        (tiene-palabra de1) (tiene-palabra la1)
                        (tiene-palabra vida1)  (patron-acentos tres_seis_diez)
                        (encabalgado-con v2e1)))

;; Apariciones de las Palabras.
(tell (:about en1 Aparicion-Palabra (de-palabra en)(anterior-a-palabra mitad1)))
(tell (:about mitad1 Aparicion-Palabra (de-palabra mitad)(anterior-a-palabra del1)))
(tell (:about del1 Aparicion-Palabra (de-palabra del)(anterior-a-palabra camino1)))
(tell (:about camino1 Aparicion-Palabra (de-palabra camino)(anterior-a-palabra de1)))
(tell (:about de1 Aparicion-Palabra (de-palabra de)(anterior-a-palabra la1)))
(tell (:about la1 Aparicion-Palabra (de-palabra la)(anterior-a-palabra vida1)))
(tell (:about v2e1 Verso  (numero-silabas-verso 11)
                        (rima-verso "ura") (tiene-palabra me1)
                        (tiene-palabra halle1) (tiene-palabra en1)
                        (tiene-palabra el1) (tiene-palabra medio1)
                        (tiene-palabra de2) (tiene-palabra una1)
                        (tiene-palabra selva1) (tiene-palabra oscura1)
                        (patron-acentos dos_seis_diez)
                        (patron-acentos cuatro_ocho_diez)
                        (encabalgado-con v3e1)))

(tell (:about me1  Aparicion-Palabra (de-palabra me)(anterior-a-palabra halle1)))
(tell (:about halle1  Aparicion-Palabra (de-palabra halle_) (anterior-a-palabra en1)))
(tell (:about en1  Aparicion-Palabra (de-palabra en)(anterior-a-palabra el1)))
(tell (:about el1  Aparicion-Palabra (de-palabra el) (anterior-a-palabra medio1)))
(tell (:about medio1  Aparicion-Palabra (de-palabra medio) (anterior-a-palabra de2)))
(tell (:about de1  Aparicion-Palabra (de-palabra de) (anterior-a-palabra una1)))
(tell (:about una1  Aparicion-Palabra (de-palabra una)(anterior-a-palabra selva1)))
(tell (:about selva1  Aparicion-Palabra (de-palabra selva)(anterior-a-palabra oscura1)))
(tell (:about oscura1  Aparicion-Palabra (de-palabra oscura)))

(tell (:about v3e1 Verso
                        (numero-silabas-verso 11)
                        (rima-verso "ida") (tiene-palabra despues1)
                        (tiene-palabra de3) (tiene-palabra dar1)
                        (tiene-palabra mi1) (tiene-palabra senda1)
                        (tiene-palabra por1) (tiene-palabra perdida1)
                        (patron-acentos dos_seis_diez)))
(tell (:about despues1 Aparicion-Palabra (de-palabra despues) (anterior-a-palabra de3)))
(tell (:about de3 Aparicion-Palabra (de-palabra de)(anterior-a-palabra dar1)))
```

```
(tell (:about dar1 Aparicion-Palabra (de-palabra dar)(anterior-a-palabra mi1)))
(tell (:about mi1 Aparicion-Palabra (de-palabra mi)(anterior-a-palabra senda1)))
(tell (:about senda1 Aparicion-Palabra (de-palabra senda)(anterior-a-palabra por1)))
(tell (:about por1 Aparicion-Palabra (de-palabra por)(anterior-a-palabra perdida1)))
(tell (:about perdida1 Aparicion-Palabra (de-palabra perdida)))

(tell (:about e1p1 Estrofa     (primer-verso v1e1)
                               (segundo-verso v2e1)
                               (tercer-verso v3e1)))

;; Como los versos son endecasilabos y riman el primero y el tercero reconocera
;; la estrofa como terceto uno tres.

(tell (:about poc1 Poema (tiene-estrofa e1p1)))
(tell (:about prc1 Prosa (tiene-linea l1pr1)))
(tell (:about l1pr1 LineaTexto
                (tiene-palabra a1) (tiene-palabra la2) (tiene-palabra mitad2)
                (tiene-palabra del2) (tiene-palabra camino2) (tiene-palabra de4)
                (tiene-palabra nuestra2) (tiene-palabra vida2) (tiene-palabra me2)
                (tiene-palabra encontre2) (tiene-palabra en2) (tiene-palabra una2)
                (tiene-palabra selva2) (tiene-palabra oscura2) (tiene-palabra por2)
                (tiene-palabra haber2) (tiene-palabra me2) (tiene-palabra apartado2)
                (tiene-palabra del2) (tiene-palabra camino2) (tiene-palabra recto2)))

(tell (:about a1 Aparicion-Palabra (de-palabra a)(anterior-a-palabra la2)))
(tell (:about la2 Aparicion-Palabra (de-palabra la)(anterior-a-palabra mitad2)))
(tell (:about mitad2 Aparicion-Palabra (de-palabra mitad)(anterior-a-palabra del2)))
(tell (:about del2 Aparicion-Palabra (de-palabra del)(anterior-a-palabra camino2)))
(tell (:about camino2 Aparicion-Palabra (de-palabra camino)(anterior-a-palabra de4)))
(tell (:about de4 Aparicion-Palabra (de-palabra de)(anterior-a-palabra nuestra2)))
(tell (:about nuestra2 Aparicion-Palabra (de-palabra nuestra)
                                         (anterior-a-palabra vida2)))
(tell (:about vida2 Aparicion-Palabra (de-palabra vida)(anterior-a-palabra me2)))
(tell (:about me2 Aparicion-Palabra (de-palabra me)(anterior-a-palabra encontre2)))
(tell (:about encontre2 Aparicion-Palabra (de-palabra encontre)
                                       (anterior-a-palabra en2)))
(tell (:about en2 Aparicion-Palabra (de-palabra en)(anterior-a-palabra una2)))
(tell (:about una2 Aparicion-Palabra (de-palabra una)(anterior-a-palabra selva2)))
(tell (:about selva2 Aparicion-Palabra (de-palabra selva)(anterior-a-palabra oscura2)))
(tell (:about oscura2 Aparicion-Palabra (de-palabra oscura)(anterior-a-palabra por2)))
(tell (:about por2 Aparicion-Palabra (de-palabra por)(anterior-a-palabra haber2)))
(tell (:about haber2 Aparicion-Palabra (de-palabra haber)(anterior-a-palabra me2)))
(tell (:about me2 Aparicion-Palabra (de-palabra me)(anterior-a-palabra apartado2)))
(tell (:about apartado2 Aparicion-Palabra (de-palabra apartado)
                                       (anterior-a-palabra del2)))
(tell (:about del2 Aparicion-Palabra (de-palabra del)(anterior-a-palabra camino2)))
(tell (:about camino2 Aparicion-Palabra (de-palabra camino)(anterior-a-palabra recto2)))
(tell (:about recto2 Aparicion-Palabra (de-palabra recto)))
```

4. Representación del vocabulario

```
 ;; Fichero Generado Automáticamente a partir de las palabras dadas por P. Gervas.
 ;; incluimos sólo un fragmento ilustrativo del total del vocabulario.
(tell (:about abandone_ Palabra_Concept
(textoPalabra "abandone_") (categoria-sintactica VLXI1S) (numero-silabas-palabra 4)
(acento 4)(rima-palabra "e_") (empVocal 1)(terVocal 1)))
```

```
(tell (:about adormecido Palabra_Concept
(textoPalabra "adormecido") (categoria-sintactica VLPPMS) (numero-silabas-palabra 5)
(acento 4)(rima-palabra "ido") (empVocal 1)(terVocal 1)))

(tell (:about amargura Palabra_Concept
(textoPalabra "amargura") (categoria-sintactica NCFS) (numero-silabas-palabra 4)
(acento 3)(rima-palabra "ura") (empVocal 1)(terVocal 1)))

(tell (:about bien Palabra_Concept
(textoPalabra "bien") (categoria-sintactica NCMS) (numero-silabas-palabra 1)
(acento 1)(rima-palabra "ien") (empVocal 0)(terVocal 0)))

(tell (:about cua_n Palabra_Concept
(textoPalabra "cua_n") (categoria-sintactica INTXMS) (numero-silabas-palabra 1)
(acento 1)(rima-palabra "ua_n") (empVocal 0)(terVocal 0)))

(tell (:about dire_ Palabra_Concept
(textoPalabra "dire_") (categoria-sintactica VLFI1S) (numero-silabas-palabra 2)
(acento 2)(rima-palabra "e_") (empVocal 0)(terVocal 1)))

(tell (:about dormido Palabra_Concept
(textoPalabra "dormido") (categoria-sintactica VLPPMS) (numero-silabas-palabra 3)
(acento 2)(rima-palabra "ido") (empVocal 0)(terVocal 1)))

(tell (:about encontraba Palabra_Concept
(textoPalabra "encontraba") (categoria-sintactica VLII1S) (numero-silabas-palabra 4)
(acento 3)(rima-palabra "aba") (empVocal 1)(terVocal 1)))

(tell (:about encontre_ Palabra_Concept
(textoPalabra "encontre_") (categoria-sintactica VLXI1S) (numero-silabas-palabra 3)
(acento 3)(rima-palabra "e_") (empVocal 1)(terVocal 1)))

(tell (:about espesa Palabra_Concept
(textoPalabra "espesa") (categoria-sintactica ADJGFS) (numero-silabas-palabra 3)
(acento 2)(rima-palabra "esa") (empVocal 1)(terVocal 1)))

(tell (:about extraviado Palabra_Concept
(textoPalabra "extraviado") (categoria-sintactica VLPPMS) (numero-silabas-palabra 4)
(acento 3)(rima-palabra "iado") (empVocal 1)(terVocal 1)))

(tell (:about fijamente Palabra_Concept
(textoPalabra "fijamente") (categoria-sintactica ADVN) (numero-silabas-palabra 4)
(acento 3)(rima-palabra "ente") (empVocal 0)(terVocal 1)))

(tell (:about grado Palabra_Concept
(textoPalabra "grado") (categoria-sintactica NCMS) (numero-silabas-palabra 2)
(acento 1)(rima-palabra "ado") (empVocal 0)(terVocal 1)))

(tell (:about hablar Palabra_Concept
(textoPalabra "hablar") (categoria-sintactica VLINF) (numero-silabas-palabra 2)
(acento 2)(rima-palabra "ar") (empVocal 0)(terVocal 0)))

(tell (:about hallaba Palabra_Concept
(textoPalabra "hallaba") (categoria-sintactica VLII1S) (numero-silabas-palabra 3)
(acento 2)(rima-palabra "aba") (empVocal 0)(terVocal 1)))

(tell (:about halle_ Palabra_Concept
```

```
(textoPalabra "halle_") (categoria-sintactica VLXI1S) (numero-silabas-palabra 2)
(acento 2)(rima-palabra "e_") (empVocal 1)(terVocal 1)))

(tell (:about inconsciente Palabra_Concept
(textoPalabra "inconsciente") (categoria-sintactica ADJGMS) (numero-silabas-palabra 4)
(acento 3)(rima-palabra "iente") (empVocal 1)(terVocal 1)))

(tell (:about mitad Palabra_Concept
(textoPalabra "mitad") (categoria-sintactica NCFS) (numero-silabas-palabra 2)
(acento 2)(rima-palabra "ad") (empVocal 0)(terVocal 0)))

(tell (:about ocurrieron Palabra_Concept
(textoPalabra "ocurrieron") (categoria-sintactica VLXI3P) (numero-silabas-palabra 4)
(acento 3)(rima-palabra "ieron") (empVocal 1)(terVocal 0)))

(tell (:about nuestra Palabra_Concept
(textoPalabra "nuestra") (categoria-sintactica ADJGFS) (numero-silabas-palabra 2)
(acento 1)(rima-palabra "uestra") (empVocal 0)(terVocal 1)))

(tell (:about pavor Palabra_Concept
(textoPalabra "pavor") (categoria-sintactica NCMS) (numero-silabas-palabra 2)
(acento 2)(rima-palabra "or") (empVocal 0)(terVocal 0)))

(tell (:about pavura Palabra_Concept
(textoPalabra "pavura") (categoria-sintactica NCFS) (numero-silabas-palabra 3)
(acento 2)(rima-palabra "ura") (empVocal 0)(terVocal 1)))

(tell (:about penoso Palabra_Concept
(textoPalabra "penoso") (categoria-sintactica ADJGMS) (numero-silabas-palabra 3)
(acento 2)(rima-palabra "oso") (empVocal 0)(terVocal 1)))

(tell (:about pensar Palabra_Concept
(textoPalabra "pensar") (categoria-sintactica VLINF) (numero-silabas-palabra 2)
(acento 2)(rima-palabra "ar") (empVocal 0)(terVocal 0)))

(tell (:about porque Palabra_Concept
(textoPalabra "porque") (categoria-sintactica CSUBX) (numero-silabas-palabra 2)
(acento 1)(rima-palabra "orque") (empVocal 0)(terVocal 1)))

(tell (:about punto Palabra_Concept
(textoPalabra "punto") (categoria-sintactica NCMS) (numero-silabas-palabra 2)
(acento 1)(rima-palabra "unto") (empVocal 0)(terVocal 1)))

(tell (:about perdida Palabra_Concept
(textoPalabra "perdida") (categoria-sintactica ADJGFS) (numero-silabas-palabra 3)
(acento 2)(rima-palabra "ida") (empVocal 0)(terVocal 1)))

(tell (:about recuerdo Palabra_Concept
(textoPalabra "recuerdo")(categoria-sintactica NCMS) (numero-silabas-palabra 3)
(acento 2)(rima-palabra "uerdo") (empVocal 0)(terVocal 1)))

(tell (:about renueva Palabra_Concept
(textoPalabra "renueva") (categoria-sintactica VLPI3S) (numero-silabas-palabra 3)
(acento 2)(rima-palabra "ueva") (empVocal 0)(terVocal 1)))
... existen en total 4872 palabras en el vocabulario que no incluimos por razones de
espacio.
```