UNIVERSIDAD COMPLUTENSE DE MADRID

Facultad de Ciencias Físicas

Departamento de Informática y Automática

UN ESTUDIO SOBRE LA APLICACION DEL RAZONAMIENTO BASADO EN CASOS A LA CONSTRUCCION DE PROGRAMAS

Inés Méndiz Noguero

Madrid, 1992

Autor: Inés Méndiz Noguero

UN ESTUDIO SOBRE LA APLICACION DEL RAZONAMIENTO BASADO EN CASOS A LA CONSTRUCCION DE PROGRAMAS

Director: Carmen Fernández Chamizo

Profesor Titular de Lenguajes y

Sistemas Informáticos

UNIVERSIDAD COMPLUTENSE DE MADRID

Facultad de Ciencias Físicas

Departamento de Informática y Automática

Año 1992

INDICE

INTRODUCCION

Existen numerosas evidencias de que los expertos en dominios tales como el derecho, matemáticas, diseño y planificación estratégica, utilizan ampliamente la experiencia adquirida en casos previos para resolver nuevos problemas. Parece lógico utilizar la misma idea en los sistemas de inteligencia artificial.

Los sistemas de razonamiento basado en casos (Case Based Reasoning, CBR) utilizan la experiencia adquirida en casos pasados para interpretar o resolver un nuevo caso. Los casos se almacenan de forma organizada en la base de conocimientos de los sistemas CBR.

Por otro lado, la Programación Automática ha sido abordada desde muy distintos enfoques. Así, la Ingeniería del Software, desde un punto de vista totalmente pragmático, intenta la reutilización automática de componentes software. Con otro enfoque, los métodos formales propugnan el desarrollo de sistemas transformacionales que partiendo de espedificaciones formales permitan la derivación de un programa. Por último, para la Inteligencia Artificial, la programación es un dominio más en el que estudirar las técincas de representación de conocimiento y razonamiento.

Centrándonos en este último enfoque, la mayoría de los sistemas "inteligentes" de síntesis de programas desarrollados hasta el momento, están basados en reglas. El objetivo central de esta tesis ha sido estudiar la aplicabilidad de los mecanismos de CBR al problema de la síntesis automática de programas. Para ello se ha desarrollado un prototipo con el que se han experimentado diversos mecanismos de inferencia y técnicas de representación del conocimiento.

La presente memoria está estructurada en cinco capítulos y un apéndice.

En el capítulo primero, analizamos desde una perspectiva histórica cuáles han sido los diversos intentos de mejorar y automatizar el desarrollo de software. En este análisis reseñamos el solapamiento a que se ha llegado entre los enfoques de la Ingeniería del Software y de la Inteligencia Artificial.

El capítulo segundo, lo hemos dedicado a los sistemas basados en el conocimiento. En él, se describen distintas técnicas de representación del conocimiento que se utilizan en Inteligencia Artificial. En particular, se describen varios sistemas de CBR y algunos sistemas de síntesis de programas.

La base de conocimientos que utilizaremos en nuestro sistema, se describe en el tercer capítulo. Se trata de una base de conocimientos dinámica en la que se almacenan los conceptos necesarios para poder establecer un diálogo con el usuario en términos del dominio de aplicación del sistema, y los casos a partir de los cuales el sistema resuelve los nuevos problemas que se le plantean. En esta biblioteca de casos, se incluyen tanto los casos

que se han dado al sistema como casos de entrenamiento, como los que ha resuelto él mismo.

El capítulo cuarto está dedicado a describir el prototipo que hemos desarrollado. En primer lugar, se exponen las estructuras de información que se suministran al sistema. A continuación, se describen los procesos mediante los cuales se incorporan esas estructuras de información a la memoria. Por último, se explican los diversos procedimientos que utiliza el sistema para obtener las nuevas soluciones.

En el capítulo quinto resumimos las aportaciones realizadas y enumeramos las posibles líneas de continuación del presente trabajo.

Finalmente, en el apéndice presentamos el listado de las funciones LISP que constituyen la implementación del prototipo.

1. PROGRAMACION AUTOMATICA Y REUTILIZACION DE SOFTWARE

1.1 INTRODUCCION

La continua expansión de la industria informática se ve frenada por las dificultades existentes en la producción de programas. Se estima que la productividad en el desarrollo de sistemas software sólo aumentó de un 3 a un 8% anual entre los años 60 y los 80, frente a más del 40% de incremento anual en la productividad de sistemas hardware durante el mismo período (Horowitz & Munson, 1984).

Simultáneamente, el coste del software se ha ido elevando. Los motivos son diversos. Podemos destacar, en primer lugar, que los sistemas de software requeridos por los usuarios son cada vez más complejos. Por otro lado, la gran demanda de personal especializado y cualificado hace que los salarios, y por tanto el coste de los productos, estén en constante alza, llegando a representar casi un 90% del coste total de las aplicaciones. El encarecimiento del software y la dificultad de disponer de programas eficientes en un plazo corto de tiempo limita la implantación de las computadoras en todas las áreas y retrasa la puesta en marcha de numerosos sistemas.

A finales de los años 60 existía, entre los profesionales de la informática, un consenso de que verdaderamente había una "crisis del software". Se llegó al convencimiento de que había que cambiar el sistema de producción de software, cuya tendencia generalizada, consistía en construir a partir de cero cada nuevo sistema de software. Un estudio realizado por Jones (1984), manifiesta que menos del 15% de todo el código escrito durante un año es realmente nuevo. El 85% restante es común con otras aplicaciones y ya ha sido programado con anterioridad.

En estas circunstancias, similares a las actuales, la investigación llevada a cabo para incrementar el desarrollo y la calidad del software constituían, y constituyen, una vía importante hacia el aumento de productividad en esta área.

Esta investigación ha sido abordada desde distintos enfoques. De ella se ha encargado tanto la Ingeniería del Software como la Inteligencia Artificial. La tarea realizada desde el punto de vista de la ingeniería del software se ha denominado Reutilización de Software (RS), mientras que se conoce como Programación Automática (PA), a los trabajos efectuados desde la perspectiva de la inteligencia artificial.

La interrelación de estas dos disciplinas -Ingeniería del software e inteligencia artificial- es un hecho desde hace varios años (Arango *et al.*, 1988; Barr & Feigenbaum, 1982; Mostow, 1985).

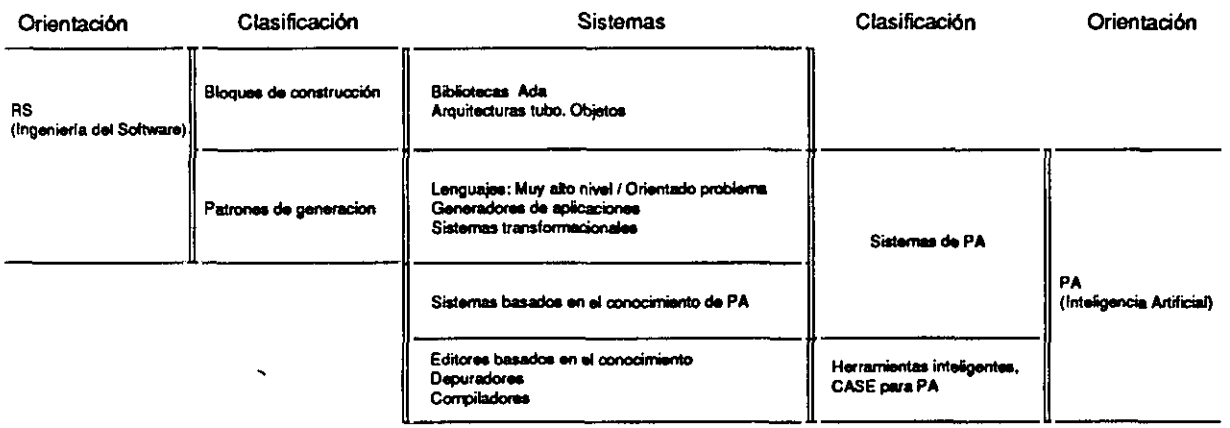

Figura 1.1

La aplicación de la inteligencia artificial a la ingeniería del software, tiene interés desde al menos dos perspectivas. Por un lado, un interés de tipo práctico para la ingeniería del software, puesto que las técnicas de inteligencia artificial mantienen la promesa de obtener mejoras, en órdenes de magnitud, de la productividad del programador y de la fiabilidad de los programas.

Por otro lado, la ingeniería del software ha demostrado ser un dominio estimulante para la inteligencia artificial. El intento de aplicar las técnicas de la inteligencia artificial a la tarea de la programación, ha motivado la investigación en la representación del conocimiento implicado y en el razonamiento automático (Rich & Waters, 1986).

Debido a esta interrelación, los mismos sistemas son considerados -por unos u otros autores- como reutilización de software o como programación automática; aunque también existen sistemas claramente definidos como de una u otra disciplina.

A continuación trazaremos unas líneas generales sobre los trabajos realizados para mejorar el desarrollo del software. Utilizaremos - tanto en la reutilización de software como en la programación automática- las clasificaciones tradicionales para enmarcar los distintos trabajos, aún con el inconveniente de citar algunos sistemas por partida doble.

En la figura 1.1, proponemos una posible clasificación de los sistemas desarrollados. La clasificación, por la propia naturaleza de la materia que estamos clasificando, no es exacta. Así, sistemas de los que hemos incluido como pertenecientes a la inteligencia artificial, pueden utilizar técnicas de ingeniería del software y vice-versa. Con esta clasificación, lo único que se pretende es dar una idea de aquellos sistemas que se consideran tanto desde la perspectiva de la reutilización de software como de la programación automática.

1.2 REUTILIZACION DE SOFTWARE

La idea de la reutilización de software está presente desde el uso de las primeras computadoras, aunque quizá no de un modo tan claro y manifiesto. Podríamos hablar de una primera reutilización elemental que consiste en cada vez que se reejecuta un programa. Otra forma consiste en la portabilidad de los programas a distintos equipos (conseguido con la aparición de los lenguajes de alto nivel y compiladores). Un modo más de reutilización son los paquetes estándares de programas que se adaptan, dentro de ciertos límites, a las necesidades propias.

Al hablar de la reutilización, sin embargo, no nos referimos a este tipo de re-utilización de programas, sino a la situación en la que se requiere un sistema, con diferencias notables a los existentes -las suficientes para que no baste una simple modificación-, y cubriendo una serie de necesidades concretas.

Como ya se ha expuesto, en términos generales, por reutilización se entiende cualquier procedimiento que produzca o ayude a producir un sistema de software partiendo de otro desarrollado anteriormente.

En un primer momento, como se verá más adelante, este proceso comprendía sólo la *reutilización del código.*

Para algunos autores, como Freeman (1983), esta reutilización debe extenderse a de todos los tipos de información que están presentes en el desarrollo del software. Sería totalmente inconsistente exhortar a los programadores para que pongan más esfuerzo en el análisis y en el diseño, y no intentar reutilizar esta información generada. Así, mientras la reutilización de una pieza de código lleva consigo, *implícitamente*, la reutilización del *análisis y del diseño, se pierde una gran oportunidad al no reutilizar explícitamente* el diseño y análisis asociado. Esta idea le lleva a introducir un nuevo término: *reusable software engineering.*

De la misma opinión es Caldiera & Basali (1991), ya que defiende que la reutilización de un objeto de software implica la reutilización concurrente de los objetos asociados a él; alrededor de un objeto, existe mucha información aunque esté de un modo informal. Así, se debe reutilizar algo más que simplemente el código.

Para otros autores, la reutilización simplemente consiste en usar un ente en un contexto diferente de aquél en el que se utilizó inicialmente (Shriver, 1987). Distingue la reutilización llamada "*black-box*" en la que el ente se utiliza tal cual, de la "*white-box*" o "*rework*" en la que, antes de utilizar esa entidad, se modifica.

A continuación vamos a ver la evolución que ha sufrido la idea y técnica de reutilización.

1.2.1 Primeras formas de reutilización de software

Una forma especializada, y primitiva, de reutilización de software, son las bibliotecas de funciones estándares. Han sido ampliamente utilizadas desde el inicio de los computadores. El éxito y valor de estas bibliotecas tradicionales de subrutinas matemáticas, que permiten al programador reutilizar funciones como el seno o la raiz cuadrada en aplicaciones muy diferentes, son bien conocidas y han servido como punto de comienzo en los primeros conceptos de reutilización.

Así, por ejemplo, ya en 1969, McIlroy hizo un claro llamamiento a la creación de componentes software, que puedan ser adaptadas y combinadas para formar nuevos programas, a semejanza de las componentes hardware de tipo estándar.

Sin embargo, la creación de simples componentes no es suficiente; no podemos esperar que todas nuestras necesidades para construir y modificar software puedan ser resueltas con una simple biblioteca de componentes. El sistema debe suministrar técnicas potentes para interconectar y modificar componentes, asi como para la catalogación y recuperación de las mismas (Goguen, 1986). Esto es lo que se empezó a realizar a partir de la segunda mitad de la década de los 70, tal como veremos más adelante.

Otro tipo de reutilización, son los llamados "Sistemas parametrizados". Este sistema de reutilización consiste en crear, para algún dominio, un sistema que lo abarque totalmente. Junto con el sistema se establece un cuestionario. Este cuestionario se rellena para cada aplicación concreta en ese dominio. Esencialmente representa cada posible opción que se puede seleccionar en un programa parametrizado. Los resultados del cuestionario se transforman en selección de opciones para un preprocesador del programa. Este preprocesador toma el sistema y lo particulariza para la aplicación deseada.

Quizá el mejor ejemplo de este método fué el realizado por IBM al final de los años 60 con el sistema llamado "Application Customizer Service". El cuestionario permitía al usuario incluir u omitir varias funciones, definir el tamaño de los campos y el formato de los resultados de los informes.

En unos estudios sobre este sistema, la experiencia reveló que la dependencia de un cuestionario, no era una buena idea. En aquellas aplicaciones en las que existen gran variedad de posibilidades, el cuestionario llega a ser rápidamente embarazoso, difícil de manejar.

Por otro lado, debido a la gran variedad de sistemas que pueden producirse, es difícil escribir una documentación adecuada. A esto se suma, que en la vida de cualquier implementación son inevitables las modificaciones de los programas añadiendo y quitando funciones. Esto no se puede realizar con un cuestionario. Es decir, a este tipo de aplicacio-

nes les falta flexibilidad. De las experiencias en este tipo de productos, podemos concluir que la reutilización en forma de sistemas altamente parametrizados se ha intentado y ha resultado ser insuficiente para el desarrollo de aplicaciones a gran escala.

A principios de los años 70, se realizaron varios desarrollos técnicos en áreas de lenguajes, estructuras de datos, sistemas operativos, transformaciones de programas, técnicas de especificación, etc., pero el principal objetivo no era el de la reutilización.

También se elaboraron estudios sobre representaciones del diseño y métodos para llevarlo a cabo (como el diseño estructurado, lenguajes de diseño de programas, diseño de Jackson, diagramas de Warnier/Orr, etc). Pero, de nuevo, todos estos adelantos no se utilizaron para realizar una efectiva reutilización.

A mediados y finales de los años 70, los profesionales del software coincidían en pensar que la reutilización del software debía ser vista como una meta, que si bien se obtendría de forma limitada por entonces, debía llegar a formas más amplias a medida que avanzara el desarrollo y la investigación.

1.2.2 Clasificación de los sistemas de reutilización de software

A finales de la década de los 70, empiezan a desarrollarse otros sistemas con una clara idea de reutilización. Siguiendo a Biggerstaff y Perlis (1984), la clasificación de las tecnologías aplicadas a la reutilización, pueden dividirse en dos grupos principales (ver figura 1.2) que dependen de la naturaleza de la componente que va a ser reutilizada: bloques de construcción reutilizables y patrones reutilizables.

Bloques de construcción reutilizables

En este primer grupo, las componentes a ser reutilizadas son bloques de construcción atómicos que son organizados y combinados de acuerdo con reglas bien definidas.

Características	Enfoques de reutilización				
Componente reutilizado	Bloques de construcción		Patrones		
Naturaleza de la componente	Atómico e inmutable. Pasivo		Difuso y maleable. Activo		
Principio de reutilización	Composición		Generación		
Enfasis	Aplicación de componente	Organización y composición de principios	Basado en el lenguaje	Generador de aplicaciones	Sistemas transformacionales
Sistemas típicos	Bibliotecas de subrutinas	Orientado al objeto Arquitecturas tubo ('pipe')	Lenguajes de muy alto nivel Lenguajes orientados al problema	Formateadores de pantallas Manejo de ficheros	

Figura 1.2

Estas componentes son atómicas en el sentido de que se mantienen invariables a través de las distintas aplicaciones y usos; su estructura interna es relativamente inmutable. Por supuesto, esta idea no siempre se consigue, y estas componentes se pueden cambiar o modificar para ajustarse a las necesidades existentes. Sin embargo, son elementos pasivos operados por un agente externo. Como ejemplos tenemos las subrutinas y funciones, programas, objetos de tipo Smalltalk, etc.

Los programas se derivan aplicando unos pocos, pero bien definidos principios de composición. Por ejemplo, el mecanismo de tubo (*pipe*) de UNIX, permite la creación de un programa complejo a partir de otros más simples conectando la salida de un programa con la entrada del siguiente. Por el contrario, los objetos de estilo Smalltalk, suministran dos principios de composición de componentes: "paso de mensajes" (una generalización de la llamada a una función), y "herencia" (que permite determinar dinámicamente, en tiempo de ejecución, la función exacta que debe ser invocada para el mensaje específico que se ha pasado a un objeto específico).

Los trabajos desarrollados pueden a su vez dividirse en dos, dependiendo de dónde se ha puesto el énfasis. Algunos investigadores se han detenido en el desarrollo y acumulación de componentes en si mismas, sin preocuparse de su organización. Otros enfocan su atención en la arquitectura general para organizar esas componentes, y las reglas generales de composición de componentes.

Los trabajos sobre *bibliotecas de componentes*, sin una mayor mayor preocupación en la especificación de las mismas y en los formalismos de composición, actualmente ya no se desarrollan. La experiencia ha demostrado que se necesitan métricas y herramientas potentes para poder desarrollar software a base de componentes. (Lanergan & Grasso, 1984; Matsumoto, 1984).

La organización y composición de componentes son los principios que rigen el segundo grupo de trabajos dentro de la reutilización de software por bloques de construcción. A través de estos principios, las componentes se combinan para formar programas. Dos ejemplos claros son el mecanismo de tubo de UNIX (Kernighan, 1984) y los lenguajes orientados a objetos. En la programación orientada a objetos, la reutilización se lleva a cabo por mecanismos distintos al tubo de UNIX. Esta orientación ha tenido, y sigue teniendo, mucho impacto en el desarrollo de software. Por este motivo, hemos dedicado un apartado para el diseño orientado al objeto más adelante.

Existen otros trabajos que son variaciones de los dos tipos citados anteriormente. Tenemos el ejemplo del realizado por Goguen (1986). Se centra en la especificación de un programa suministrando tres mecanismos (*theories*, *views* y *images*) que permiten que las

especificaciones sean divididas en pequeñas estructuras de modo que se aumente su reutilización.

Como se verá más adelante, en estos enfoques a base de componentes, existen diversos problemas como la clasificación e identificación de las componentes, modificación de las mismas en los casos que sea necesario. Estos y otros problemas se estudiarán más detenidamente.

Reutilización de patrones

Este otro grupo de sistemas que incorporan la reutilización no es fácil de caracterizar. En este caso, las componentes son elementos activos que generan el programa. Su reutilización es más por ejecución que por manipulación.

En algunos sistemas, como los generadores de aplicaciones, las componentes reutilizables pueden ser patrones de código en el propio generador. En otros, como los sistemas de transformaciones, las componentes pueden ser patrones que existen en un conjunto de reglas de transformación.

En los dos casos, las componentes individuales de reutilización, son difíciles de caracterizar e identificar, debido a que sus efectos en el entorno del programa tienden a ser más globales y difusos que en la reutilización por bloques de construcción. Las estructuras resultantes a menudo admiten sólo relaciones distantes con los patrones a partir de los cuales fueron generados.

De forma semejante, no es posible caracterizar los principios por los cuales las componentes son reutilizadas. El sentido de reutilización está menos definido que en el grupo de bloques de construcción. Aquí, las componentes son reutilizadas en el mismo sentido en que un programa se reutiliza cuando se reejecuta. Esta clase de reutilización es una reactivación de un mecanismo de generación; es decir, una reactivación de un patrón que dirige la generación.

Dentro de la reutilización de patrones, se pueden distinguir tres clases de actividades, de nuevo dependiendo de dónde se pone el énfasis: sistemas basados en el lenguaje, generadores de aplicaciones y sistemas transformacionales. No existe un límite claro en esta división, de forma que un sistema concreto puede pertenecer a más de una de estas clases.

Los sistemas basados en el lenguaje, son aquellos sistemas de generación en los que el principal énfasis se hace en la notación que se va a utilizar para describir el sistema. Estos sistemas cubren un amplio espectro desde los lenguajes de propósito general (a menudo llamados Lenguajes de Muy Alto Nivel -*Very High Level Languages*, VHLL-) a Lenguajes de Propósito Especial (comúnmente llamados lenguajes orientados al problema -*Problem Oriented Languages*, POL-). La característica clave que diferencia estos tipos de lenguajes,

es el grado en el que los constructores del lenguaje tienen información específica del dominio problema. (Cheng *et al.*, 1984).

Los generadores de aplicaciones suelen codificar mucha información específica del dominio, pero -en su mayor parte- incorporada en el propio programa del generador. La entrada a estos sistemas rara vez se establecerse como un lenguaje; o bien el usuario la suministra interactivamente, o bien es una entrada de texto altamente limitado sin un conjunto rico de constructores.

Los generadores de interfaces de pantallas son un ejemplo típico de generadores. El usuario describe la estructura de la pantalla para un programa creándola "*on-line*" a base de códigos especiales o secuencias claves para describir los campos que forman la pantalla. Los sistemas de manejo de ficheros para ordenadores personales, utilizan unos descriptores -concisos, pero limitados- para establecer los campos, que se podrían considerar como un lenguaje. Quizá uno de los mayores éxitos de la programación automática comercial ha sido precisamente el desarrollo de generadores de bases de datos (Rich & Waters, 1988). Estos sistemas son limitados y no son convenientes para aplicaciones complejas. Sin embargo, permiten a los usuarios construir sus propias bases de datos y acceder a las mismas sin ayuda de programadores.

El objetivo de los sistemas transformacionales es permitir especificar una aplicación en una notación concisa, fácil de comprender. El trabajo de los sistemas transformativos es reescribir esa especificación de entrada en una forma ejecutable eficiente (y también como contrapartida, más oscura). Estos sistemas realizan cambios incrementalmente en el entorno del programa a través de reglas explícitas. Estas reglas están separadas del sistema transformacional, es decir, están representadas como datos. Los sistemas transformacionales en si mismos, son muy simples, examinan el entorno del programa y buscan patrones de emparejamiento para una o más reglas. Si encuentran una regla aplicable, la aplican. El proceso continua hasta que se encuentra una condición de terminación (Broy, 1983).

1.2.3 Opciones en la reutilización

En el problema de la reutilización, influyen distintos factores, planteándose diversas alternativas. Respecto a éstas, de forma general, puede decirse que un cambio positivo en un factor, produce, a menudo, un cambio negativo en otro. Los factores a tener en cuenta son los siguientes:

a) Generalidad de aplicación frente a rentabilidad. Las tecnologías que son muy generales, es decir, que pueden aplicarse a una extensa gama de dominios, obtienen un resultado menos potente que aquellos sistemas que se centran en uno o dos dominios de aplicación.

Por ejemplo, si un generador de aplicaciones facilita la creación de interfaces de pantallas y bases de datos, las facilidades disponible por el usuario son mucho mayores que si se utiliza un lenguaje de alto nivel de propósito general ya que tiene pocas piezas de arquitectura específicas para construir interfaces de pantalla y bases de datos.

b) Tamaño de la componente frente a su potencial de reutilización. Cuanto mayor es una componente mayor es el resultado que produce, pero, al mismo tiempo, también es más específica: el dominio de aplicación se reduce, y dificulta la reutilización incrementan - do el coste de la misma cuando sean necesarias las modificaciones.

Este dilema se amplifica cuando lo que se intenta reutilizar es el código, ya que éste, por su propia naturaleza, es muy especificaco.

c) Coste de la biblioteca. El tercer dilema con el que nos encontramos es que se debe invertir un gran capital intelectual, económico y de tiempo antes de obtener resultados significativos. Generalmente, el desarrollo de software se lleva a cabo a través de distintos grupos de personas trabajando en diversos proyectos. Dichos proyectos tienen un presupuesto y unas metas. Estas metas, generalmente, no cuentan con un trabajo extra para generalizar y obtener los resultados del proyecto que puedan ser reutilizados por otros grupos.

1.2.4 Problemas operacionales

Al establecer la clasificación de sistemas de reutilización, hablamos de componentes, bien pasivas o activas. La idea de resolver un nuevo problema a base de interconectar componentes, parece sencilla de llevar a cabo, pero cuando se realiza de forma práctica, nos encontramos con una serie de problemas que podríamos llamar operacionales.

A grandes rasgos, podemos hablar de cuatro problemas operacionales:

a) Búsqueda de componentes. El proceso de localización de una componente es mucho más complicado que la simple localización de un emparejamiento (*matching*) exacto. Este proceso debe incluir la localización de componentes muy similares, ya que lo que se necesitará en la mayoría de los casos es reutilizar parcialmente una componente, más que en su totalidad.

Las posibilidades que se han propuesto al respecto han sido variadas. Prieto-Díaz & & Freeman (Prieto-Díaz & Freeman, 1987; Prieto-Díaz, 1991) ha desarrollado un esquema de clasificación para una biblioteca de componentes; Por otro lado, el sistema Paris (Katz *et al.*, 1987) añade precondiciones y poscondiciones a las propiedades que debe verificar la componente para realizar la búsqueda. El resultado es una lista de componentes candidatas.

b) Comprensión de las componentes. El proceso de comprensión de la componente se requiere tanto si se necesita modificarla como si no, pero especialmente, si debe modificarse. El usuario de la componente necesita un modelo mental de la computación de la componente para usarla adecuadamente. Cada modelo mental supone el esfuerzo de adquirirlo. Este es seguramente, el problema operacional fundamental que debe resolverse en el desarrollo de cualquier sistema de reutilización con independencia de la tecnología escogida para su implementación.

c) Modificación de componentes. La modificación de las componentes es el alma de la reutilización. Esta cambia la vision de un sistema de reutilización desde una biblioteca estática a un sistema de componentes con vida que produce, cambia y desarrolla nuevas componentes cuando se modifican los requisitos.

Existen pocas herramientas que suministren ayuda para la modificación de las componentes.

Bassett (1987) propone un marco de trabajo basado en *frames*. Cada *frame* es una "solución modelo" a una clase de problemas de programación relacionados. Este modelo contiene unos puntos, ya especificados, en los que se encuentran los cambios dependiendo de cada problema específico. Estos *frames* permiten formalizar las propiedades comunes que comparten una serie de programas, de forma que el computador puede producir las *instances* (casos particulares) para cada problema específico. El código de esas características específicas lo aportarían otros *frames*.

d) Composición de componentes. La composición de componentes introduce uno de los requisitos más desafiantes en la representación usada para especificar las componentes. Esta representación debe tener una doble característica: Debe suministrar la posibilidad de representar estructuras compuestas como entidades independientes, con unas características computacionales bien definidas. Y también dar la posibilidad de componer estas estructuras compuestas para formar nuevas estructuras computacionales con un conjunto de características computacionales diferentes.

1.2.5 Lenguajes de diseño orientado al objeto y reutilización

Hace años, la creación de bibliotecas de componentes software estaba dificultada por la carencia de tipos abstractos de datos. La mayoría de los lenguajes de programación tenían una visión muy simple de los tipos de datos, centrándose principalmente en los tipos numéricos; ello obligaba a describir cualquier otro tipo de dato en términos numéricos, estableciendo correspondencias entre el tipo abstracto deseado y un tipo numérico. Esto hacía que no hubiera acuerdo en cuanto a las correspondencias que se requerían, ya que situaciones diferentes puedían tratarse mejor con correspondencias diferentes, y estas diferencias se extiendían a todo el programa; normalmente, un cambio en la correponden-

cia usada, requiería la reescritura completa del programa. Esa disparidad de opiniones era lo que impedía la creación de bibliotecas para tipos de datos no numéricos (Barnes, 1982).

En los primeros días de la informática los lenguajes ensambladores facilitaban a los programadores la utilización de las instrucciones máquina (operadores) para manipular los elementos de datos (operandos). El nivel de abstracción que se aplicaba al dominio de la solución era muy bajo.

Conforme aparecieron los lenguajes de programación de alto nivel (por ejemplo FORTRAN, ALGOL, COBOL), los objetos y las operaciones del espacio del problema del mundo real podían ser modelados mediante datos y estructuras de control predefinidas, que estaban incorporadas en el lenguaje de alto nivel. En general, el diseño de software (siempre que se considerara explícitamente) se enfocaba sobre los detalles de la representación procedimental de acuerdo con el lenguaje de programación elegido. Los conceptos de diseño, tales como refinamientos sucesivos de una función, modularidad procedimental y, posteriormente, programación estructurada, fueron introducidos entonces.

Durante los años 1970, se introdujeron conceptos tales como la abstracción y el ocultamiento de información, y emergieron métodos de diseño conducidos por los datos, pero los que desarrollaban software aún se concentraban, principalmente, en el proceso y su representación. Al mismo tiempo, los lenguajes de alto nivel modernos (por ejemplo, Pascal) introdujeron una variedad mucho más rica de tipos y estructuras de datos.

Al mismo tiempo que los lenguajes de alto nivel convencionales (lenguajes a partir de FORTRAN y ALGOL) evolucionaban durante los años 60 y 70, los investigadores centraron sus esfuerzos en una nueva clase de lenguajes de simulación y de construcción de prototipos, tales como SIMULA y Smalltalk. En estos lenguajes, la abstracción de datos tenía una gran importancia, y los problemas del mundo real se representaban mediante un conjunto de *objetos*, a los cuales se les añadía el correspondiente conjunto de *operaciones*. El uso de estos lenguajes era radicalmente diferente del uso de los lenguajes más convencionales.

Durante los años 1980 la rápida evolución de los lenguajes de programación como Ada o Smalltalk, causaron un creciente interés en el Diseño Orientado al Objeto (DOO). En las primeras discusiones sobre el método para conseguir un diseño orientado al objeto, Abbott (1983) mostró cómo el análisis de la descripción del problema y su solución en lenguaje natural, puede usarse como guía para desarrollar la parte visible de un paquete útil (un paquete que tenga los datos y procedimientos que operan sobre ellos) y el algoritmo particular para un problema dado. Booch (1983) extendió el trabajo de Abbott y ayudó a popularizar el concepto de diseño orientado al objeto.

El diseño orientado al objeto, como otras metodologías de diseño orientadas a la información, crea una representación del dominio del problema en el mundo real y lo transforma en un dominio de solución a nivel software. A diferencia de otros métodos, el DOO da como resultado un diseño que encapsula los objetos de datos (elementos de datos) y las operaciones de procesamiento, de forma que modulariza la información y el procesamiento en vez de sólo el procesamiento.

Wiener y Sincovec (1984) resumen la metodología DOO de la siguiente manera: "Ya no es necesario para el diseñador de sistemas convertir el dominio del problema en estructuras de datos y control predefinidas, presentes en el lenguaje de implementación. En vez de ello, el diseñador puede crear sus propios tipos de abstracción de datos y abstracciones funcionales y transformar el dominio del mundo real en estas abstracciones creadas por el programador. Esta transformación, incidentalmente, puede ser mucho más natural debido al virtualmente ilimitado rango de tipos abstractos que pueden ser inventados por el diseñador. Además, el diseño del software deja aparte los detalles de implementación de los objetos de datos usados en el sistema. Estos detalles de representación pueden cambiarse muchas veces, sin que se produzcan efectos inducidos en el sistema de software global".

Más adelante veremos las aportaciones del DOO a la reutilización. A continuación vamos a exponer el papel que ha jugado Ada en el DOO y en la reutilización.

Ada: Exitos y limitaciones

Ada representa un caso particular en el enfoque de "objetos" suministra constructores para soportar tanto la abstracción de datos como la de programas (Buzzard & Mudge, 1985), aunque no la herencia, proporcionando un entorno adecuado para la programación basada en objetos.

A continuación, vamos s describir brevemente algunas de sus características.

Abstracción de datos. Los constructores que implementan los tipos abstractos de datos son los tipos *package* (paquete) y *private.* El paquete de Ada coloca con efectividad un muro alrededor de un grupo de declaraciones de forma que permite acceder sólo a aquellas declaraciones que son intencionadamente visibles. Los paquetes tienen dos partes: la declaración y el cuerpo (*body*). La declaración del paquete especifica formalmente los tipos abstractos de datos y su interfaz con el mundo externo. El cuerpo del paquete contiene ocultos los detalles de implementación.

Abstracción de programas. La abstracción de programas permite operaciones sobre objetos implícitos. Así, además de ocultar la representación y acceso de un objeto, también

se oculta su existencia. El resultado es una forma más completa de ocultamiento y, normalmente, una interfaz más concisa que la abstracción de datos.

La abstracción de programas en Ada se realiza a través del *generic package* (paquete genérico). Ada permite la declaración de unidades genéricas de programas que sirven como plantillas para paquetes o subprogramas a partir de los cuales se pueden obtener los paquetes o subprogramas necesarios. Las unidades genéricas de programa pueden tener parámetros actuales que suministran los detalles específicos de la particularización de la plantilla.

Las unidades genéricas permiten un nivel más alto de abstracción que la abstracción de datos, ya que los tipos abstractos de datos pueden esconderse totalmente en un caso particular (*instance*) del cuerpo del paquete genérico. La ocultación de la estructura de datos se lleva a cabo a través de las variables internas del paquete que no son locales a los programas del paquete.

La mayor diferencia entre la abstracción de datos y de programas radica en cuál es la unidad de programa que posee el del tipo abstracto de datos. En la abstracción de datos, el tipo es poseido por una unidad de programa que es externa a la unidad que maneja el tipo. En la abstracción de programas, el tipo está en la unidad que lo maneja. En Smalltalk, la abstracción ha sido dispuesta de tal manera que todos los objetos se tratan igual, con independencia de si ese objeto representa módulos de programas o estructuras de datos. (Goldberg & Robson, 1983)

Las posibles aportaciones de los tipos abstractos de datos en este campo, así como las ventajas que aporta la utilización de lenguajes de programación tipo Ada, donde un programa está diseñado como una colección de grandes componentes software (Pratt, 1986), han sido ampliamente reconocidas (Gargaro & Papas, 1987). Aunque también se ha puesto claramente de manifiesto que un lenguaje por sí sólo no puede resolver el problema de la reutilización (Tracz, 1988).

Diseño orientado al objeto

Junto con esta técnica de diseño, aparecen nuevos conceptos propios del diseño orientado a objetos: objetos, clases y herencia, que describiremos a continuación.

De estos conceptos, solo uno, la herencia, es una contribución única de esta metodología. La ligazón de esta herencia con los otros es lo que caracteriza específicamente la programación orientada a objetos.

Los objetos. Un objeto es una componente del mundo real que se implementa en el dominio del software. Por ejemplo, objetos típicos pueden ser: máquinas, órdenes, archivos,

visualizaciones, conmutadores, señales, cadenas alfanuméricas o cualquier otra persona, lugar o cosa.

Los objetos toman espacio de la memoria y tienen una dirección asociada como un registro en Pascal o una estructura en C. La disposición de los bits en el espacio de memoria ocupado por un objeto, determina el estado de ese objeto en cualquier momento dado.

Asociado con cada objeto, hay un conjunto de procedimientos y funciones, llamados *operaciones* o *métodos*, que definen las operaciones significativas sobre ese objeto. Un *mensaje* es una petición al objeto para que ejecute una de sus operaciones (Pressman, 1987).

Un objeto, pues, encapsula tanto el estado como el comportamiento.

Clases. Una clase define un conjunto de posibles objetos. Desde el punto de vista de un lenguaje de fuerte tipificación, una clase es un constructor para implementar un tipo definido por el usuario. Idealmente, una clase es una implementación de un tipo abstracto de datos. Esto significa que los detalles de implementación de una clase son privados a dicha clase. La interfaz pública de esta clase se compone de dos tipos de métodos. El primer tipo consiste en funciones de acceso que devuelven abstracciones significativas sobre el estado de un caso particular. El otro tipo de método se refiere a procedimientos de trasformación para pasar un caso particular de un estado válido a otro. La idea de ocultamiento de la información conduce a que todos los datos en una clase deben ser privados. Esto garantiza que la interfaz de una clase es, de hecho, una abstracción.

Herencia. Herencia es una relación entre clases que permite definir e implementar una clase en función de otra ya existente. La herencia es el concepto más prometedor que puede ayudarnos a construir sistemas software a partir de partes reutilizables. La abstracción procedimental funciona bien en algunos dominios concretos como las bibliotecas matemáticas, pero la unidad abstraída es demasiado pequeña, el enfoque procedimental no es suficientemente general y los mecanismos de parámetros son demasiado rígidos. Otro mecanismo que ha sido presentado como una esperanza clave en la reutilización es el concepto de tipos parametrizados o genéricos aplicados a procedimientos y paquetes de Ada. La capacidad de crear bibliotecas de procedimientos genéricos y paquetes es muy útil, pero aún no ha sido suficientemente explotada y contituirá un aspecto fundamental en futuros desarrollos de software. Sin embargo, sólo es aplicable a sistemas fuertemente tipificados y no es un concepto general como la herencia.

Soporte para la reutilización

Este diseño orientado al objeto combina las técnicas de diseño y las características del lenguaje para suministrar un buen soporte para la reutilización de módulos software. Esta reutilización puede realizarse de varias formas. Algunas de estas técnicas, en el diseño

orientado al objeto son casi iguales que en lenguajes procedimentales, pero añaden otros tipos de reutilización (Korson & McGregor, 1990).

Cada vez que se crea un caso particular ("instancia") de una clase, existe reutilización. Esto es similar a la declaración de una variable de un tipo específico. La principal diferencia es que el resultado de particularizar una clase es una estructura mucho más compleja que una simple variable. Una instancia de una clase suministra una combinación de estructuras de datos, y operaciones sobre esas estructuras.

Declarar un caso particular de una clase **string** para declarar el atributo **nombre** en una clase **empleado** es un ejemplo de este tipo de reutilización. Usar una particularización de la clase string suministra operaciones para copiar, concatenar, y si es necesario, editar el valor del string.

La herencia porporciona dos niveles de apoyo para la reutilización. Como parte de una fase de diseño de alto nivel, la herencia sirve como un medio de modelización de las relaciones generalización/especialización. Estas relaciones aparecen en forma de clasificaciones. Una silla puede ser vista como un tipo especial de mobiliario, así como una descripción más general de categorías más específicas como silla plegable o silla giratoria, por ejemplo. Este uso de la herencia a alto nivel fomenta el desarrollo de abstracciones significativas, lo que, a su vez, favorece la reutilización.

En el diseño de software tradicional las abstracciones a nivel medio, como mesa o silla, se reconocen y consideran separadamente. La disponibilidad de una relación de herencia permite al diseñador identificar las partes comunes entre las abstracciones y producir abstracciones de un nivel más alto -por ejemplo mobiliario- a partir de estas partes comunes. A través de esta identificación de partes comunes y de llevarlas a un nivel más alto de abstracción, llegan a estar disponibles para que puedan ser reutilizadas más tarde en el diseño real y en diseños sobre mobiliarios. Muchas de estas descripciones (atributos como altura, anchura, color, etc) pueden estar ya disponibles desde la abstracción de mobiliario. Los beneficios de esta reutilización animan al diseñador a buscar niveles de abstracción más altos.

En la fase de diseño a bajo nivel, la herencia soporta la reutilización de una clase existente como base para la definición de una nueva clase. Una pieza de código puede copiarse a un nuevo fichero y modificarse para encajar en el nuevo propósito. Esta "herencia en la edición", no establece ninguna relación entre la nueva y vieja pieza de código. Si se descubre un fallo en el código viejo y se repara, el conocimiento de este cambio puede o no llegar a la persona responsable del código copiado. La herencia aporta una importante mejora en este aspecto.

La herencia establece una dependencia entre la clase existente y la nueva. El código heredado se incluye en la nueva definición cuando la definición de la clase nueva se compila. Cualquier modificación en la clase original -reparar errores o añadir características- se incorporan a la nueva clase en la próxima compilación. Esta técnica permite que una clase sirva como base para muchas nuevas definiciones sin propagar los errores de las definiciones originales a través del sistema.

1.3 PROGRAMACION AUTOMATICA

Programar una computadora consiste en especificar las distintas tareas que debe efectuar la computadora para alcanzar un objetivo. Esta especificación debe expresarse en términos que la computadora pueda enterder. Cuando es otro programa el que establece esas tareas, el proceso se denomina programación automática (Biermann, 1987).

La que se acaba de introducir, es una definición simple de lo que se entiende por programación automática. Los investigadores en este campo no han adoptado una definición única, sino que existen diversas definiciones dependiendo del énfasis dado a las distintas facetas.

El objetivo último de la programación automática es conseguir un sistema que pueda mantener un diálogo en lenguaje natural con un usuario no programador acerca de los requisitos de la aplicación a implementar y que, a continuación produzca un programa adecuado.

Como corresponde a un objetivo tan a largo plazo, mucho de lo que originariamente se concibió como programación automática, se ha conseguido hace ya años. Por ejemplo, los primeros lenguajes ensambladores y compiladores, fueron considerados como sistemas de Programación Automática (Cfr. ACM Vol. 1, No. 4, April 1958, p. 8).

Por el momento, los objetivos inmediatos de la programación automática se centran fundamentalmente en intentar dar una solución a la denominada "crisis del software".

La investigación actual pretende conseguir herramientas que permitan la máxima automatización posible en la construcción de software, a fin de disminuir los costes y los tiempos de desarrollo. Simultáneamente se pretende aumentar la fiabilidad de los programas, en orden a garantizar que todo el software cumpla las especificaciones para las que fue construido. Por otro lado, el coste del mantenimiento de un programa representa una parte muy significativa en el coste total del ciclo de vida del software. Un sistema de programación automática que permita la rápida modificación y adaptación del software a nuevas especificaciones, producirá un notable ahorro en los costes de mantenimiento.

La programación automática, como campo de investigación bien definido tal y como se entiende en la actualidad, comenzó al principio de los años 70. Un trabajo de Manna y Waldinger (1971) introdujo la terminología y sugirió un método de síntesis de programas recursivos e iterativos. A partir de entonces, gran cantidad de trabajos se han desarrollado utilizando muy diversas técnicas.

Desde un principio, y con frecuencia, se ha sobreestimado la capacidad de los sistemas de programación automática. Los principales errores que se suelen cometer son los siguientes:

a) Se proclama que los sistemas de programación automática están orientados al usuario final y no necesitan tener conocimientos del dominio. La realidad es que estos sistemas deben ser expertos en el dominio. En la conversación humana, muchas cosas se sobreentienden por el contexto y por el uso de palabras propias del tema. Para poder detallar brevemente las especificaciones de un programa, es necesario que el oyente conozca los vocablos específicos del dominio para evitar, al que habla, la necesidad de definirlos. Estas ideas se resumen en la frase: *El comportamiento inteligente, requiere conocimiento* (Mostow, 1985).

b) Se postula la viabilidad de un sistema orientado al usuario final, de propósito general y completamente automático. Dada la imposibilidad práctica de soportar simultáneamente estas tres características, los sistemas que se desarrollan, se centran en dos de ellas a costa de prescindir de la tercera (Rich & Waters, 1988). Las tres posibles vias son las siguientes:

i.- *Bottom up* (De abajo a arriba). Sacrifica la orientación al usuario final, de forma que el usuario de estos sistemas debe ser un especialista. Se ha pasado de lenguajes máquina a lenguajes de Alto Nivel (LAN) y las investigaciones actuales se dirigen a lenguajes de muy alto nivel (LMAN).

ii.- *Narrow domain.* (Dominio reducido). Sacrifica el propósito general. Centrándose en dominios más reducidos, se puede llegar a conseguir generadores de programas totalmente automáticos que se comuniquen directamente con el usuario final. La investigación sobre este tipo de sistemas está dirigida a cubrir dominios más amplios.

iii.- *Assistant.* (Ayudante). Sacrifica la total automatización. Son entornos de programación consistentes en colecciones de herramientas como editores inteligentes, documentación de ayuda *on-line*, y analizador de programas. La meta aquí está en mejorar la integración entre herramientas y el nivel de asistencia proporcionado por cada herramienta individualmente.

c) Otro error que se comete frecuentemente es la suposición de que la programación es un proceso en serie, en el que la fase de especificación, realizada por el usuario, no se entremezcla con la fase de implementación desarrollada por el programador. La programación es un proceso interactivo, caracterizado por el continuo diálogo entre el usuario final y el programador, ya que, normalmente, es difícil para el cliente establecer explícitamente, al principio, todos sus requisitos (Pressman, 1987).

Herramientas CASE

Como ya se ha esbozado en los párrafos anteriores, la inteligencia artificial ofrece dos posibilidades de automatización del software:

- Herramientas inteligentes de desarrollo de software (*Assistant*). Programas de inteligencia artificial específicamente diseñados para ayudar al programador a realizar las distintas fases del desarrollo de software.

- Programación automática. Programas de inteligencia artificial para desarrollar otros programas de acuerdo con las especificaciones del programador (objetivo último de la programación automática).

Los programadores usan programas especiales, herramientas CASE (*Computer Assistant Software Engineering*), para incrementar su productividad en las distintas etapas del proceso de desarrollo de software. Estas herramientas también se podrían considerar como parte del trabajo desarrollado desde el punto de vista de la ingeniería del software (MIS-1).

El proyecto *Aprendiz de programador* (Rich, 1981; Rich & Waters, 1988) utiliza algunas de estas herramientas. Entre ellas está un editor basado en el conocimiento (Waters, 1981, 1982). Incluye conocimiento sobre un lenguaje específico de ordenador y sobre programación en general. Puede identificar y corregir errores de sintaxis que sean introducidos por el programador antes de que provoquen problemas en la ejecución del programa; esto es posible por su conocimiento sobre el lenguaje de programación. Y debido a que posee conocimientos de programación, puede generar partes del programa en respuesta a pequeñas especificaciones hechas por el programador.

1.3.1 Clasificación de los sistemas de Programación Automática

En este apartado, nos centraremos en los sistemas de programación automática propiamente dichos.

Todos los sistemas de programación automática tienen un objetivo común: automatizar la programación. Sin embargo existen diversos puntos en los que se diferencian. Podríamos decir que los dos más característicos son el tipo de especificaciones que admite

el sistema, es decir, la forma en que se plantea el problema a resolver; y el método de obtención del programa a partir de las especificaciones .

Tipos de especificaciones

Desde el punto de vista del usuario, el aspecto más sobresaliente de la programación automática es el lenguaje que debe utilizar para comunicarse con el sistema.

Un programa es un producto cuyos requisitos deben estar perfectamente establecidos. De la precisión en las especificaciones depende que el programa satisfaga o no las necesidades de sus usuarios.

La importancia de las especificaciones ha sido recalcada frecuentemente; pero si importante es el papel de las especificaciones en la programación convencional, en la programación automática su papel es vital, ya que, en este caso, representa el puente de comunicación entre el hombre y la computadora. No es concebible un buen sistema de síntesis de programas si no se dispone de un buen método de especificación.

A grandes rasgos, podríamos distinguir cinco tipos de especificación:

Lenguaje natural. Los lenguajes naturales son una alternativa atractiva para la comunicación entre el usuario y el sistema de programación automática. Las tres características que presentan al lenguaje natural como una opción interesante son: el vocabulario, la informalidad y la sintaxis.

El vocabulario, porque existen ya miles de palabras predefinidas lo que hace del lenguaje natural un medio eficiente de comunicación.

La informalidad (por ejemplo la posibilidad de que las sentencias sean ambiaguas, incompletas, incluso contradictorias) también es importante. De hecho, muchas veces, para describir algo complejo, se comienza con una pequeña descripción e incrementalmente se va *modificando hasta que es aceptable.*

La sintaxis es la característica menos importante del lenguaje natural. La conveniencia de esta sintaxis es simplemente que resulta familiar a los que ya están habituados a ese lenguaje.

Desafortunadamente, por el momento no es posible desarrollar máquinas que puedan establecer un diálogo en lenguaje natural, utilizando técnicas actuales de la inteligencia artificial. Sería necesario elegir un subconjunto de dicho lenguaje y, según muchos expertos, el aprendizaje de un subconjunto sería más difícil que el de un lenguaje formal, a causa de la interferencia con los hábitos de uso del lenguaje natural.

Por otra parte, las especificaciones en lenguaje natural no pueden ser verificadas automáticamente por una computadora para determinar su consistencia.

Sin embargo, la investigación en la comunicación del hombre con el computador por medio del lenguaje natural (sea o no utilizado para la programacin automática) es plenamente actual (Barnett *et al.*, 1990). Según Abbott (1987), el problema de la representación del conocimiento (crucial en la programación automática como veremos en el próximo capítulo) no estará resuelto hasta que tengamos un sistema de representación razonablemente completo para el lenguaje natural. En este sentido, se están desarrollando diveros sistemas como IPP (Lebowitz, 1980) o DMAP (Direct Memory Access Parsing).

El primer proyecto en esta línea, dentro de la programación automática, fue el de Heidorn (1972). La finalidad de este proyecto era desarrollar un sistema que, después de mantener un diálogo en inglés, acerca de un problema simple de colas, generara un programa de simulación en lenguaje GPSS. Se construyó un sistema procesador de lenguaje natural (NLP) de propósito general, y este sistema se usó para desarrollar el sistema de programación automática para problemas de colas (NLPQ), proporcionándole una gramática apropiada e información sobre colas.

Otro proyecto de esta misma época fue el desarrollado por Balzer (1972). Se concebía que un sistema de programación automática tenía cuatro fases: adquisición del problema, transformación, verificación del modelo y codificación automática. La primera fase constituía en un diálogo en lenguaje natural.

Posteriormente se ha estado trabajando en el paso de especificaciones informales a formales. Uno de los trabajos más actuales es SPECIFIER (Miriyala & Harandi, 1991). Es sistema un sistema interactivo que deriva especificaciones formales de tipos de datos y programas a partir de sus descripciones informales. El proceso de derivar una especificación formal se enfocq como un proceso de resolución de problemas, de tal forma que utiliza esquemas, analogía y razonamiento basado en diferencias.

Lenguajes de propósito especial. Aun cuando las personas hablan, no siempre eligen el lenguaje natural como medio de comunicación. Por ejemplo, en muchas áreas hay lenguajes gráficos o símbolos especializados asociados al dominio concreto (por ejemplo las fórmulas matemáticas) que es lo que utilizan los expertos. Son los lenguajes denominados de propósito especial.

Estos lenguajes estarían muy cercanos a los lenguajes orientados al problema (POL), tratados al hablar de la reutilización de software.

Existen muchas clases de lenguajes de propósito especial que pueden soportar dominios suficientemente estrechos (*Narrow Domain*). Un éxito particular han tenido las

interfaces llamadas WYSWYG (*"What you see is what you get"*). Este tipo de interfaces por pantalla permite a los usuarios indicar su propósito. El generador escribe automáticamente el código de acuerdo con lo que se le ha indicado en la pantalla, y consultando en su base de datos.

El problema fundamental de estos lenguajes es el campo, tan limitado, de aplicación. Esto plantea un problema aún no solucionado: cómo combinar uno o varios lenguajes de propósito especial con otro de propósito general.

Ejemplos. Una idea atractiva, propuesta en los primeros años de la programación automática, consiste en especificar un programa a través de ejemplos de su comportamiento. Lo interesante de este enfoque es que los no-programadores están acostumbrados a los ejemplos como técnica de comunicación (tanto como a los lenguajes naturales o de propósito especial). Además, las colecciones de ejemplos son fáciles de comprender y modificar.

Excepto para problemas triviales, no se han conseguido contruir sistemas de programación automática basados en ejemplos. Lo que se pretende con este tipo de especificación, es que el sistema maneje los datos de entrada de forma "análoga" a los ejemplos. La experiencia ha demostrado, que con independencia de la cantidad de ejemplos que se proporcionen al sistema, no existe la seguridad de que la abstracción que derive -para aplicarla posteriormente- sea correcta.

El primer trabajo en este campo fue el de Biermann (1972) que se refiere a la inferencia de máquinas de Turing a partir de ejemplos de cálculo. Otro ejemplo es el sistema de Smith (1982) que estudia las máquinas de inferencia inductiva (MII), capaces de sintetizar programas a partir de ejemplos de entrada-salida. Estas máquinas son dispositivos algorítmicos que toman como entrada el grafo de una función de los números naturales en los números naturales, y que producen como salida el programa que sintetiza dicha función.

Formalismos lógicos. La lógica es el lenguaje de descripción formal conocido más potente y general. Como resultado, es razonable suponer que debería ser un buen medio de comunicación entre el usuario y el sistema de programación automática.

Sin embargo, existen dos obstáculos para utilizar los formalismos lógicos. Primero, que las tareas más interesantes de la lógica formal -como detectar contradicciones-, son computacionalmente intratables. Segundo, la complejidad de las fórmulas lógicas hace que sean notoriamente difíciles de escribir y comprender para la mayoría de la gente.

La investigación sobre la lógica como medio de comunicación entre el hombre y la máquina, está llevándose a cabo bajo los temas de lenguajes de especificación formal y

lenguajes de programación lógica. Una cuestión clave en estas dos áreas es la introducción de extensiones y restricciones que hagan a la lógica más tratable al hombre. Por ejemplo Prolog (Cohen, 1985) garantiza la ejecutabilidad de una descripción lógica imponiendo fuertes restricciones en la forma de las expresiones.

Lenguajes de muy alto nivel. Los lenguajes de muy alto nivel se construyen a partir de los actuales lenguajes de alto nivel. Normalmente, estos lenguajes añaden la posibilidad de utilizar tipos abstractos de datos para evitar a los programadores los detalles de la implementación de las estructura de datos, y algunas pocas características de notación lógica, para que los programadores puedan ignorar ciertas clases de detalles de los algoritmos. De nuevo, este tipo de especificación de la entrada, es común -como ya vimos- a la investigación en reutilización de software.

Entre los lenguajes de muy alto nivel, podemos citar el lenguaje SETL (Goldberg, 1983) que está basado en los axiomas básicos de la teoría de conjuntos. Este lenguaje permite que los algoritmos sean programados rápidamente sin requerir que se declaren las estructuras de datos. Tales declaraciones se pueden especificar posteriormente a mano, sin tener que recodificar el programa, para mejorar la eficiencia en la ejecución del programa.

Otro lenguaje de muy alto nivel es el lenguaje BDL (Hammer *et al.*, 1977) cuyo dominio son las aplicaciones mercantiles. BDL es un lenguaje basado en flujo de datos (*data-flow language*) y por ello no dispone de primitivas de flujo de control sino que una operación se realiza cuando todas sus entradas han sido calculadas.

Más allá de los temas discutidos hasta aquí, los siguientes tres principios deben ser garantizados por cualquier medio de comunicación que se desee como apto para los sistemas de programación automática:

Primero, un medio debe ser de amplio espectro. El usuario debe poder especificar todo, desde propiedades muy abstractas, hasta consejos sobre detalles de implementación a bajo nivel. Esto es necesario porque los sistemas de programación automática no pueden operar sin obtener cierta cantidad de consejos a todos los niveles.

Segundo, debido a la naturaleza inherentemente interactiva de la programación, el medio debe ser capaz de soportar un diálogo entre el usuario y el sistema de programación automática.

Tercero, el medio debería tener un gran vocabulario de términos predefinidos de forma que el sistema pueda conversar con el usuario a un nivel deseable, para evitar que el usuario tenga que definir los términos que va utilizando.

Métodos de síntesis

Los sistemas de programación automática buscan la obtención de un programa (una descripción en términos específicos de implementación) a partir de una descripción en términos específicos del dominio (en uno de los medios de entrada descritos en el apartado anterior). Actualmente, se proponen cuatro mecanismos para realizar esta tarea: métodos procedimentales, deductivos, transformacionales y de inspección.

Métodos procedimentales. Hasta la fecha, el enfoque de mayor éxito ha sido simplemente, escribir un programa de propósito especial que obtiene los resultados adecuados. Por ejemplo, la mayoría de los compiladores y de los generadores de programas son esencialmente procedimentales por naturaleza, aunque unos pocos utilicen transformaciones.

La gran ventaja de estos métodos es que pueden construirse rápidamente, ya que se pueden proyectar como sistemas para soportar una serie limitada de características. Siempre se puede modificar el código para soportar características adicionales.

Sin embargo, a medida que más y más características se añaden al sistema procedimental, se llega a un punto en el que el sistema llega a ser progresivamente cada vez más difícil de modificar; no tienen la capacidad de ampliación (*scalability*). Como resultado, es improbable que este enfoque procedimental pueda soportar, en un futuro, sistemas de programación automática, orientados al usuario final cubriendo un espectro razonable de dominios.

Métodos deductivos. El problema de sintetizar un programa satisfaciendo una especificación dada es formalmente equivalente a buscar una demostración constructiva de que cumple la espeficación. Esta idea fundamental subyace en el enfoque deductivo. En principio, cualquier método automático de deducción -resolución, deducción natural- pueden usarse para soportar la programación automática.

Desafortunadamente, en la práctica ninguno de estos métodos está aún disponible para demostrar la clase de teoremas complejos requeridos para sintetizar programas de tamaño real.

La deducción es básicamente un problema de búsqueda por un camino de inferencia, desde algún conjunto inicial de hechos a uno final. La búsqueda es exponencial por naturaleza, ya que para cada paso existen muchos modos en los que las reglas de inferencia se pueden aplicar a los hechos. Los sistemas de deducción automática actuales no pueden obtener demostracciones profundas porque no son capaces de controlar con efectividad el control sobre el proceso de búsqueda.

Para tratar este problema de control, los sistemas deductivos, normalmente toman un enfoque de asistente, es decir, piden consejos al usuario. Sin embargo, los usuarios que

quieren evitar la programación, no suelen estar dispuestos a guiar la demostración de un teorema.

Un problema más fundamental de los enfoques deductivos es que estos buscan simplemente una demostración, pero no buscan la eficiencia.

A pesar de estas limitaciones, los métodos deductivos tienen ciertas ventajas, como por ejemplo el hecho de que son de propósito general - aunque limitados a problemas simples-. Como resultado, podemos decir que los métodos deductivos jugarán un papel importante en el futuro de la programación automática. Para ello será necesario combinar la deducción automática con otros métodos, de forma que se puedan evitar sus limitaciones inherentes.

Los primeros tabajos en este campo fueron los de Green (1969), y Lee *et al.* (1974). Los sistemas desarrollados producían programas condicionales, pero su capacidad de construcción de ciclos era muy rudimentaria.

Posteriormente destacan los trabajos de Manna & Waldinger (1979), que desarrollaron el sistema DEDALUS. Este sistema aceptaba especificaciones, dadas en lenguaje de alto nivel, que no contenían ninguna indicación acerca del algoritmo que debía emplearse. A continuación, el sistema intentaba transformar las especificaciones en un programa recursivo escrito en lenguaje tipo LISP, utilizando para ello aproximadamente un centenar de reglas de transformación.

Métodos transformacionales. Muy abundantes en la investigación actual de programación automática y en la reutilización del software. En este enfoque, la entrada al sistema de programación automática es un programa escrito en un lenguaje de muy alto nivel. Posteriormente se aplican un conjunto de transformaciones para convertir esa entrada en una implementación a bajo nivel.

Una transformación tiene tres partes: un patrón, un conjunto de condiciones lógicas de aplicabilidad y un procedimiento que se activa cuando se satisfacen dichas condiciones. Cuando se encuentra una *instance* de un patrón, se chequean las condiciones de aplicabilidad para ver si la transformación se puede aplicar. Si se satisfacen las condiciones de aplicabilidad, se evalua la acción para computar una nueva sección de código, que se usa para reemplazar al código emparejado (*matching*) por el patrón. Normalmente, las transformaciones preservan la corrección.

Existen dos clases básicas de transformaciones. Algunas transformaciones reemplazan construcciones a nivel de especificación (por ejemplo una cuantificación sobre un conjunto) por estructuras convencionales (por ejemplo, la iteración sobre una lista). Estas transformaciones codifican conocimiento de cómo implementar algoritmos y estructuras

de datos. Otras transformaciones realizan adaptaciones y optimizaciones (por ejemplo poner fuera de un bucle una operación invariante), que no cambian el nivel de abstracción. En la práctica, estas dos clases de transformaciones están entrelazadas en largas secuencias, pasando a través de multiples niveles de abstracción.

La característica central de los métodos transformacionales es el *ciclo de reescritura transformacional*. El estado del proceso de una transformación se representa como un programa en una representación de amplio espectro capaz de representar tanto la entrada del usuario como el resultado deseado. En cada ciclo, un sistema transformacional selecciona una transformación y la aplica sobre algún lugar del programa. El ciclo continua, acumulando los resultados en una cadena cada vez más larga de transformaciones, hasta que alguna condición se satisface (por ejemplo, hasta que no hay más constructores de muy alto nivel).

En muchos casos, no existe mucha diferencia entre las secuencias de pasos de transformaciones y la secuencia de pasos de demostración. Por lo tanto, no sorprende que los sistemas transformacionales sufran el mismo problema de control que los métodos deductivos. Como consecuencia, los métodos transformacionales deben también o pedir consejos al usuario, o imponer fuertes restricciones en las clases de transformaciones que pueden ser usadas.

Los sistemas transformacionales que piden consejos no son mucho más satisfactorios que los deductivos que también los piden y solo se han realizado en el laboratorio. Sin embargo, los modulos transformacionales restrictivos pueden encontrarse como componentes de varios compiladores y en otros sistemas.

Un aspecto interesante de las secuencias de transformación es que normalmente contienen un pequeño porcentaje de pasos claves (normalmente tomando decisiones de cómo implementar abstracciones) entrelazados con muchos pasos pequeños, menos intuitivos, que establecen y determinan aspectos alrededor de auéllos clave. La investigación actual sobre estos métodos se dirige directamente a la automatización de estos pequeños pasos mientras se pide consejo al usuario sobre los pasos clave.

La mayor fuerza de estos métodos transformacionales consiste en que suministran una clara representación de ciertas clases de conocimiento de programación. Por esta razón los métodos transformacionales, de alguna forma, son válidos para ser parte de futuros sistemas de programación automática.

Dos claros ejemplos de sistemas transformacionales son PECOS (Barstow, 1979) y LIBRA (Kant & Barstow, 1981). Los dos son subsistemas del PSI y utilizan reglas de transformación. PECOS es el codificador y LIBRA un experto en eficiencia.

Métodos de inspección. Los programadores humanos, rara vez piensan sólo en términos de elementos primitivos como la asignación y los tests. Al contrario, como los ingenieros de otras disciplinas, normalmente piensan en términos de combinaciones de clichés o esquemas de elementos correspondientes a conceptos que les son familiares.

Conociendo un número suficiente de clichés, es posible realizar muchas tareas de programación por inspección en vez de por razonamiento a partir de los primeros principios. En la síntesis por inspección, el paso de la especificación a la implementación se realiza por medio de clichés: primero se reconoce un cliché que encaja con la especificación, y a continuación, se escoge un cliché de implementación dependiendo del cliché obtenido para la especificación. Usando la comprensión global, los métodos de inspección reducen el problema de control de búsqueda que se presenta en otros métodos.

La característica central de los métodos de inspección es la codificación y uso de clichés. Un cliché tiene tres partes: un esqueleto que está presente en todas las ocurrencias de ese cliché, *roles* cuyos contenidos varian de un caso a otro y unas ligaduras (*constraints*) sobre lo que puede llenar esos *roles*. Una propiedad esencial de los clichés es su interrelación. Por ejemplo un cliché puede especializar o extender otro cliché. En el Aprendiz de programador (Rich, 1981; Rich & Waters, 1988) la codificación de los clichés y el razonamiento sobre éstos es la actividad central del sistema.

Actualmente se está trabajando mucho sobre estos métodos. Se hace incapié en la conocimiento acumulado por los expertos y la *analogía* que éstos establecen entre el problema que se les propone y los que ya han solucionado anteriormente. Como ejemplos, tenemos los sistemas APU (Bhansali, 1991) y SPECIFIER (Miriyala & Harandi, 1991) que comentaremos en el capítulo segundo.

El sistema que presentamos en el capítulo cuarto, se encuentra dentro de este último grupo de métodos de síntesis. Utiliza la analogía con casos que tiene en su memoria, para resolver un nuevo problema.

En este capítulo hemos descrito los distintos trabajos que se han realizado para intentar dar una solución a la denominada "crisis del software". Por un lado están los trabajos realizados desde el punto de vista de la ingeniería del software, y por otro los realizados desde el punto de vista de la inteligencia artificial.

Uno de los resultados a los que se llegó es al papel crucial que juega el conocimiento del dominio sobre el que trabaja el sistema. Poco a poco también se fue poniendo de manifiesto la necesidad de que los sistemas sean capaces de aprender, con el paso del tiempo, de su propia experiencia.

En el próximo capítulo se da una visión general de los sistemas basados en el conocimiento, deteniéndonos con mayor detalle en los sistemas de Razonamento Basado en Casos (CBR), dentro de los cuales se encuadra el sistema que hemos desarrollado.

2. SISTEMAS BASADOS EN EL CONOCIMIENTO

2.1 INTRODUCCION

La inteligencia artificial es una ciencia de creciente interés interdisciplinario y práctico. Numerosos profesionales trabajando en áreas muy dispares han desarrollado nuevas ideas y herramientas en esta ciencia. Así, psicólogos teóricos han desarrollado nuevos modelos de la mente basados en conceptos fundamentales de la inteligencia artificial (sistemas de símbolos y procesamiento de la información). Los lingüistas también están interesados en estas nociones básicas así como en trabajos de inteligencia artificial en lingüística computacional. Por su parte, los informáticos están usando modelos mentales para mejorar el comportamiento de sus sistemas.

La inteligencia artificial como ciencia moderna empieza seriamente en los años 50. En el verano de 1956, en la conferencia de Dartmouth, recibió su nombre. Sin embargo, debido al gran campo que abarca, los investigadores no se ponen muy de acuerdo sobre una única definición para inteligencia artificial.

No es inusual encontrar científicos que se consideran a sí mismos trabajando en el campo de la inteligencia artificial, pero no son considerados de esta forma por alguno de sus colegas. Por el contrario, hay científicos trabajando en áreas que "tradicionalmente" se consideran parte de inteligencia artificial y, no obstante, rehúsan aplicar ese término a su trabajo.

Así tenemos definiciones informales como la que recoge Mostow (1985), según la cual, lo que la inteligencia artificial pretende es que las máquinas hagan cosas que, en opinión de la gente, requieren inteligencia. Según Barstow (1981), inteligencia artificial es la parte de la informática que se dedica al diseño de sistemas de computadoras inteligentes, es decir, que exhiban las características que nosotros asociamos con la inteligencia en el comportamiento humano -comprensión del lenguaje natural, aprendizaje, razonamiento, resolución de problemas, etc-. Por último, Rich (1987) la define como el estudio de las formas -modos- en las cuales puede construirse un sistema para realizar tareas cognitivas que -en el momento actual- la gente realiza mejor.

En todas ellas se habla de un resultado final -el comportamiento inteligente- con independencia del proceso interno que ejecuta la computadora. De tal forma, que en los primeros problemas abordados por la inteligencia artificial al comienzo de los años 60, el objetivo prioritario estaba encaminado hacia la eficiencia computacional. Se buscaba sólo un resultado final inteligente aunque su proceso no tuviera nada que ver con el proceso cognitivo que sigue una persona humana. Así tenemos por ejemplo el GPS (*General Solver Problem*) (Ernst & Newell, 1969), STRIPS (un solucionador general de tareas para el robot SHAKEY del SRI) (Fikes & Nilson, 1971), el juego del ajedrez, etc.

Los primeros resultados fueron estimulantes y, en esta euforia, Newel y Simon hicieron su famosa predicción de 1958: antes de 1968 será campeón de ajedrez un programa y se demostrará un importante teorema matemático. Pero hay que saber que, en realidad, estos primeros programas eran de tipo combinatorio, es decir, ensayaban muchas posibilidades sin examinar ampliamente ninguna de ellas. Shannon hizo una estimación considerando que, aun si el computador pudiera evaluar un millón de movimientos por segundo, tardaría 10 elevado a 95 años en seleccionar un movimiento. Tales procedimientos permiten obtener resultados aceptables en numerosos campos, pero muy raramente es posible obtener muy buenos resultados. Para obtenerlos mejores habría que examinar todavía más combinaciones, pero en este caso -como ya hemos visto- los tiempos de cálculo aumentaban muy deprisa con una mejora muy lenta de la calidad del resultado.

La gran aportación de estos primeros sistemas fue una idea importante: el papel clave que juega el conocimiento en el control de todo el proceso.

Esta idea señala directamente el motivo por el que esos primeros sistemas no eran viables. Se debía a su carencia de conocimiento. En otras palabras, esos sistemas explotaban un mecanismo de resolución general, pero este mecanismo accedía a muy poca cantidad de conocimiento del dominio sobre el cual trabajaban.

El objetivo prioritario se decantó para intentar lograr que los procesos y algoritmos utilizados por los programas simulasen el comportamiento humano. Los trabajos de Newell *et al.* (1960) y Quillian (1967) fueron progresivamente abriendo paso a una concepción de la inteligencia artificial en la que la simulación cognitiva ocupaba un puesto destacado.

La investigación de la inteligencia artificial entre los años 70 y primeros de los 80, permitió el desarrollo de la primera generación de los sistemas basados en el conocimiento. El resultado más visible fue la introducción de los "sistemas expertos": sistemas software que encierran la suficiente competencia como para realizar tareas que normalmente se reservan a los expertos humanos.

Los sistemas expertos de esa época se basaban en reglas (y de hecho, muchos autores identifican sistemas expertos con sistemas basados en reglas). Obtuvieron gran auge y proliferación. Por ejemplo el sistema DENDRAL (Buchanan & Feigenbaum, 1978) que realiza hipótesis acerca de las estructuras moleculares, partiendo de la espectrografía de masas; el sistema MYCIN (Davis *et al.*, 1977) para diagnosticar enfermedades infecciosas de la sangre y recomendar un tratamiento; y así muchos otros (Buchanan, 1983).

Dentro de la programación automática se construyó el sistema PSI (Green, 1977; Barstow, 1979). Permitía escribir programas utilizando lenguaje natural, pares de entrada-salida y trazas parciales, y producía implementaciones eficientes de dichos programas en lenguaje LISP o SAIL. Entre otros, tenía dos subsistemas, el codificador (PECOS) (Bars-

tow, 1979) y un experto en eficiencia (LIBRA) (Kant & Barstow, 1981). El codificador sugería las implementaciones posibles y LIBRA seleccionaba las más eficientes, teniendo también en cuenta el coste de los recursos que tiene que utilizar el sistema para desarrollar tales implementaciones.

Se desarrollaron numerosas herramientas para la construcción de sistemas expertos, pero, a pesar de este auge, pronto se pusieron de manifiesto sus limitaciones: estos sistema no soportaban los complejos requisitos necesarios para las futuras aplicaciones a gran escala. Mark & Simpson (1991) señalan cuatro características de las que carecían:

a) *scalability* (capacidad de ampliación). Las herramientas estaban orientadas a crear sistemas relativamente pequeños, con serias limitaciones en su nivel y rango de experiencia.

b) Respuesta en tiempo real. Las herramientas y arquitecturas no podían soportar sistemas expertos que pudieran reaccionar rápidamente o suministrar una respuesta con las suficientes garantías en un intervalo de tiempo limitado.

c) Reusalibidad. El conocimiento extraído, con dificultad, para resolver un problema particular no era fácil de reutilizar en otros problemas.

d) Metodologías para la integración. Era difícil integrar metodologías diferentes -sistemas de múltiple representación o técnicas de razonamiento, por ejemplo- para tratar problemas complejos.

Simultáneamente con el desarrollo de estos sistemas expertos, en la segunda mitad de los 70, Minsky (1975) propuso un nuevo formalismo para representar el conocimiento. Se trataba de los *frames* o marcos. Dos años más tarde, Schank y Abelson (1977) proponían los *scripts* o guiones. Los dos formalismos se basaban en la idea de que existe abundante experiencia psicológica de que las personas utilizan grandes cantidades de conocimiento, *bien organizado*, obtenido en experiencias previas para interpretar las nuevas. Los *frames* son unas estructuras (un marco de trabajo) donde están los objetos y eventos *típicos* de una situación específica. Los *scripts* son estructuras como los *frames* específicamente diseñadas para representar secuencias de eventos. Suponía un paso importante en la representación del conocimiento. Empezaron así lo que podríamos llamar sistemas basados en esquemas (Anderson, 1983)

De forma general, al principio de la década de los 80, se reconoció que los expertos humanos aplican más conocimientos, y conocimientos de más tipos, que los que esos primeros sistemas basados en reglas podían acomodar.

Entre 1985 y 1990, se avanza significativamente, tanto en la cantidad y tipos de conocimientos que se pueden aplicar a los problemas, como en las herramientas y métodos

que soportan la construcción y ejecución de sistemas basados en el conocimiento. El software basado en el conocimiento, ahora puede:

a) Representar y razonar con modelos detallados de dispositivos complejos o sistemas software.

b) Seleccionar o generar un modelo apropiado, basado en el problema a resolver que se está tratando.

c) Usar la experiencia de resolución de problemas previa del sistema como guía para resolver los nuevos problemas (Mark & Simpson, 1991).

Estos avances se han conseguido principalmente por dos tipos de sistemas expertos que han emergido en los últimos 5 años: los sistemas basados en modelos y los sistemas basados en casos (Milne, 1991; DARPA, 1989).

Los sistemas basados en modelos son especialmente útiles en el diagnóstico de averías de equipos. En contraposición a los sistemas basados en reglas, que están sustentados en la experiencia humana, los basados en modelos lo están en el conocimiento de la estructura y conducta de los dispositivos que intentan "entender". En efecto, un sistema experto basado en modelos incluye un "modelo" de un dispositivo que puede usarse para identificar las causas de los fallos del mismo. Debido a que los sistemas basados en modelos sacan conclusiones directamente del conocimiento de las estructura y conducta del dispositivo, se dice que razonan a partir de los "primeros principios".

Una caracterísica muy atractiva de los sistemas basados en modelos es su "transportabilidad". Los sistemas basados en reglas que incorporan el conocimiento de un experto sobre diagnóstico de problemas de equipos en un computador determinado, pueden no tener ningún valor cuando se trata de reparar un computador diferente. Por el contrario, si se pudiera desarrollar un sistema experto que incluyera el conocimiento de cómo trabajan los circuitos electrónicos digitales de un computador, teóricamente podría ser usado para diagnosticar problemas que se presenten en cualquier computador.

Como ejemplo de estos sistemas tenemos el desarrollado por Val & Morueco (1989) sobre diagnosis y verificación de diseño de cirtuitos electrónicos. De acuerdo con las ideas de "diagnosis por primeros principios", la diagnosis se realiza mediante el análisis de las discrepancias entre valores calculados a partir del modelo y los valores observados. La diagnosis se refina iterativamente mediante observaciones adicionales. La modelación utilizada permite el tratamiento de circuitos a distintos niveles de descomposición, lo que da lugar a una diagnosis por niveles jerárquicos.

CYRAH (Cuena *et al.*, 1989) es un sistema experto que integra el razonamiento simbólico basado en reglas y dos modelos matemáticos de simulación cuantitativa, es decir,

se trata de un sistema mixto con razonamiento basado en reglas y razonamiento basado en modelos.

Los otros sistemas que han surgido en estos últimos años son los sistemas basados en casos. El sistema que hemos desarrollado utiliza esta metodología, por lo que la expondremos con más detalle, después de presentar la teoría cognitiva que subyace en este tipo de sistemas.

2.2 ENFOQUE COGNITIVO DE LA INGENIERIA DEL SOFTWARE

Tal como se dijo anteriormente, la investigación actual en inteligencia artificial tiende a simular el comportamiento humano. Así, el estudio del comportamiento de los programadores humanos puede ofrecer una guía útil para mejorar la automatización de los sistemas. Este enfoque ha sido aplicado a un gran número de sintetizadores de programas (Barstow, 1979; Rich & Waters, 1988; Bhansali, 1991; Steier, 1991).

A continuación discutiremos los dos principios básicos subyacientes en nuestro trabajo: el comportamiento de los programadores humanos y el papel que juegan los mecanismos de analogía y aprendizaje en el diseño de software.

2.2.1 El comportamiento de los programadores humanos

Muchos investigadores han analizado el comportamiento de los estudiantes de informática, de los posgraduados y de los programadores expertos mientras trabajan en problemas de diseño de algoritmos. (Soloway & Ehrlich, 1984; Kant, 1985). Los resultados de algunos de estos estudios se resumen en (Steier, 1991) y se exponen a continuación:

- Los diseñadores tienen representaciones mentales abstractas de algoritmos y fragmentos de algoritmos que no están ligadas a un lenguaje de programación en particular.

- Normalmente, los diseñadores eligen, al principio, un esquema básico para el diseño del algoritmo, y el resto del tiempo lo gastan en refinar ese esquema inicial.

- La estrategia de control usada en el diseño del algoritmo es una estrategia de refinamiento *top-down*, en la que se estudia el objetivo que se quiere alcanzar, el estado en el que nos encontramos, y como consecuencia se ponen los medios necesarios para alcanzar ese fin (análisis de fines y medios, o *means-end analysis*).

- Existe una continua ejecución mental del algoritmo parcialmente desarrollado a fin de permitir el análisis *means-end*.

- El análisis del comportamiento de los programadores demuestra la evidencia de una profundización progresiva, similar a la que se encuentra en el análisis de protocolos de los jugadores de ajedrez.

- Los diseñadores usan la explicación (a otros o a ellos mismos) de cómo funciona el algoritmo para detectar errores.

- El aprendizaje es esencial en todo el proceso del diseño. Los humanos continuamente aprenden. El aprendizaje se usa para facilitar la construcción de programas similares y para adquirir nuevas terminologías y estrategias de diseño.

Otros investigadores han estudiado el papel del lenguaje de programación en el comportamiento de los programadores (Davies, 1991). La conclusión de estos estudios muestra que en los programadores de Pascal (independientemente de su nivel de experiencia), existen -y son significativos- ciertos esquemas mentales que guian la programación, mientras que para los programadores de BASIC (con poca experiencia o experiencia media), sólo existen algunas pistas sobre el control de flujo en los programas a la hora de diseñar nuevos programas.

Parece ser que en los primeros niveles de destreza en la programación, los aspectos de la notación de Pascal soportan el uso de planes, pero en un nivel de destreza suficientemente elevado, el efecto de la notación llega a ser menos importante. Davies sugiere que la notación del lenguaje de programación y la representación mental del conocimiento interaccionan fuertemente para determinar la estrategia de programación. Esta suposición explicaría el comportamiento diferente entre programadores de Pascal y de BASIC en niveles de destreza baja o intermedia.

2.2.2 Analogía, CBR y aprendizaje

Como se ha dicho anteriormente, los programadores humanos realizan muchas tareas de programación por inspección de esquemas mentales previamente adquiridos más que por razonamiento a partir de los primeros principios (Rich & Waters, 1988; Steier, 1991). Estos métodos de síntesis de programas por inspección últimamente están basados en la experiencia. Además, automatizar la construcción de programas por inspección requiere la utilización de técnicas de representación y manejo del conocimiento y, obviamente, exige capacidad de aprendizaje por parte del sistema.

Como expresa Dershowitz (1986) la analogía es una herramienta que los sistemas de programación automática pueden utilizar para aprender a partir de la experiencia, tal como hacen los programadores. Una analogía entre la especificación de un programa dado y la de un nuevo problema puede usarse como base para modificar dicho programa de modo que se adecúe a la nueva especificación.

El enfoque del razonamiento basado en casos (CBR) intenta imitar esta característica del pensamiento humano. Se ha propuesto como un modelo psicológicamente más plausible que el razonamiento basado en reglas, que es la base de muchos sistemas expertos.

Sería muy difícil para un programador experto producir un conjunto de reglas que justifiquen su comportamiento en la resolución general de problemas. Sin embargo, podría fácilmente resolver problemas particulares explicando cada paso del proceso -incluyendo referencias a conocimiento estático o heurísticas que ha usado- junto al porqué utilizó ese razonamento. Este proceso suministra unas trazas de derivación externas que una ingeniería de inferencia analógica puede utilizar para resolver problemas similares futuros de forma efectiva. Así, el conocimiento de un caso, expande -tanto si han sido casos de experiencias pasadas como adquiridos externamente- la habilidad de resolver más y más problemas.

Existen razones por las cuales el enfoque del CBR se establece como un avance en el aprendizaje por parte del sistema (DARPA, 1989). La adquisición del conocimiento es mucho más fácil en el CBR que en otros métodos de aprendizaje, los cuales, en general, necesitan tener grandes cantidades de conocimiento disponible antes de que el proceso de aprendizaje pueda ser útil. La razón es que mucho de este conocimiento, en el CBR, está en forma de casos. Los casos necesitan una depuración mínima de la interacción entre ellos. Así, la adquisición del conocimiento inicial puede ser "maquinal". Además, en muchos dominios, existen casos bases que pueden ser usados como "semillas" de un sistema basado en casos.

En este trabajo, proponemos el CBR como una aproximación natural del problema del diseño de software. Gran parte de la enseñanza de la programación se realiza a través de ejemplos. A partir de un conjunto de programas de entrenamiento, los estudiantes pueden abstraer esquemas o técnicas generales de programación. Este mismo mecanismo puede ser utilizado en sistemas de síntesis de programas.

2.3 SISTEMAS BASADOS EN CASOS

Pensamos que dentro de los sistemas basados en el conocimiento, el formalismo que mejor se adapta al estudio de la programación automática, tal como hemos visto en el apartado anterior, es el Razonamiento Basado en Casos ("Case-Based Reasoning", CBR) (Riesbeck & Schank, 1989). Somos de la opinión de que cuando un programador resuelve un problema, utiliza su experiencia pasada -problemas ya resueltos y comprobados- para construir el nuevo programa.

Para introducir el tema vamos a empezar con un ejemplo en el que comparamos el razonamiento basado en reglas (RBR) frente el razonamiento basado en casos, CBR.

Supongamos que mostramos un sistema de canales y depósitos a un estudiante de Física y a un encargado de riego de fincas. Les hacemos preguntas sobre cómo se comportaría el flujo del agua al abrir distintas compuertas. El estudiante de Física, probablemente, resolverá el problema utilizando razonamiento basado en reglas (fórmulas), mientras que el encargado de riego posiblemente usará CBR. El estudiante tardará varios minutos, quizá tenga algunas equivocaciones pero finalmente hallará la solución exacta y podrá afrontar cualquier otro problema similar. El encargado de riego dará una respuesta aproximada casi instantáneamente, pero no una respuesta exacta y seguramente tendrá problemas cuando se enfrente con situaciones inusuales.

El modo en que un razonador basado en casos resuelve los problemas consiste en buscar en la memoria del sistema aquellos casos que resuelven problemas similares al actual -teniendo en cuenta cualquier diferencia entre la situación actual y la anterior- y adaptar la o las soluciones de esos casos de la memoria para que se acomoden al caso actual.

El objetivo de seleccionar casos relevantes de la memoria de casos es recuperar aquellos casos a través de los cuales se podrían hacer predicciones sobre el nuevo. La recuperación se hace utilizando las características del nuevo caso que fueron relevantes en la solución de casos pasados. Las soluciones anteriores han sido indexadas por estas características relevantes para asegurar el éxito de la recuperación, lo que significa que la estructura de la memoria de casos juega un papel principal en este proceso.

Un sistema basado en reglas será flexible y producirá respuestas casi óptimas, pero *será lento y propenso a errores. Un sistema* CBR *estará restringido a las variaciones sobre* situaciones conocidas y producirá respuestas aproximadas, pero será rápido y sus respuestas serán establecidas como experiencia real. En dominios muy limitados, los sistemas basados en reglas darían mejores resultados, pero la situación se invierte cuando los dominios se hacen más realísticamente complejos. En cuanto trabajemos con un dominio complejo, necesitaremos un montón de reglas, muchas de las cuales serán sutiles y difíciles de verificar, y las cadenas de razonamiento serán largas. Con CBR siempre hay una conexión corta entre el caso de entrada y la solución recuperada.

Según Riesbeck & Schank (1989), el CBR ofrece dos ventajas sobre el RBR.

1) La experiencia de los expertos, es mas similar a una biblioteca de experiencias pasadas que a un conjunto de reglas; de aquí que los casos soportan mejor la transferencia de conocimiento (comunicación de la experiencia de los expertos del dominio al sistema) y la explicación (justificación de la solución dada desde el sistema a los expertos del dominio).

2) Muchos dominios en el mundo real son tan complejos que o bien es imposible o bien no es práctico especificar completamente todas las reglas involucradas; sin embargo, siempre se pueden dar casos, es decir soluciones a problemas concretos.

2.3.1 Razonamiento basado en casos

El ciclo básico de un sistema de CBR es:

1.- Introducción del problema actual,

2.- Buscar una solución anterior relevante y

3.- Adaptar esa solución al caso actual.

El primer problema consiste en determinar qué viejas situaciones son similares a la actual. Las viejas soluciones relevantes tienen que haber sido etiquetadas de forma que las características de los problemas nuevos puedan ser usadas para encontrarlas. Pero, normalmente, la relevancia no puede ser establecida por las características obvias del problema de entrada, sino por relaciones abstractas entre características, ausencia de características, etc.

Cuando se introduce un problema en un sistema CBR, la fase de análisis determina las características relevantes para encontrar casos similares. Estas características se suelen llamar *índices* en la literatura del CBR. El problema de la *indexación* consiste en determinar qué características extra, no obvias y no presentes en la entrada son necesarias en un dominio particular.

Normalmente, a través de los índices, se recupera un conjunto de casos anteriores potencialmente relevantes. El siguiente paso es comparar (*matching*), otra vez, esos casos con la entrada. Así, se rechazan los casos que son demasiado distintos, y se determina cuál de los restantes es el más similar al caso de entrada. La similitud entre casos depende de cómo emparejen entre ellos en cada dimensión -cada índice-, y de lo importante que sea esa dimensión.

Después de haber elegido el caso que mejor se corresponde con el actual hay que adaptarlo a la nueva situación. El proceso de adaptación consta de dos fases: determinar las diferencias entre el caso nuevo y el viejo y modificar la solución almacenada para el caso viejo teniendo en cuenta esas diferencias. Las reglas necesarias para este proceso son complejas y difíciles de caracterizar de forma general. Posteriormente se discutirán distintas técnicas de adaptación.

La "cantidad" de adaptación necesaria depende de la naturaleza de las diferencias entre los dos casos -actual y anterior-. A veces, lo que se hizo para el caso recuperado, servirá para la nueva situación. En otras ocasiones, simplemente se necesitarán pequeñas modifi-

caciones. Si hay tantas diferencias entre el caso nuevo y el caso recuperado que el sistema es incapaz de adaptar la solución, lo mejor es que el sistema pida a un experto humano que resuelva el problema y guarde dicha solución en la biblioteca de casos.

2.3.2 Organización de la memoria

La memoria del sistema de CBR que vamos a utilizar -se explicará con más detalle en capítulos posteriores- está organizada utilizando MOPs ("Memory Organization Packages") (Schank, 1982).

Los sistemas que utilizan memorias basadas en MOPs incluyen nociones estándares de la inteligencia artificial como *frames*, abstracción, herencia, etc., pero aplicadas -en estos sistemas- a una base de conocimientos que cambia dinámicamente, es decir, que aprende nuevos conocimientos en los procesos de comprensión y resolución de los problemas.

Memory Organization Packages (MOPs)

La unidad básica en una memoria dinámica es el MOP. Un MOP se usa para representar conocimiento sobre clases de sucesos, especialmente sucesos complejos. Un MOP contiene un conjunto de normas (*norms*) que representan las características básicas del MOP (eventos, objetivos, actores, etc.). Los MOPs, básicamente, contienen la misma información que los *scripts* que proponían Schank y Abelson (1977), pero los *scripts* eran estáticos mientras que los MOPs son estructuras dinámicas, cambian con el uso. Además los MOPs se organizan en redes entrelazadas.

Un MOP puede tener especializaciones, es decir MOPs que son versiones mas específicas de ese MOP. El MOP que representa una ocurrencia particular de un suceso se llama caso particular (*instance*). Un caso -de los almacenados en la memoria de un sistema CBR- puede referirse tanto a una *instance*, como a un MOP más general.

Los MOPs de una memoria dinámica están unidos por enlaces (*links*). Un sistema de CBR sigue esos enlaces para obtener, a partir de un MOP, el siguiente. Estos enlaces juegan un papel clave en el razonamiento basado en casos porque determinan qué información está disponible al razonador y en qué momento.

En general, pueden utilizarse enlaces de diversos tipos:

1.- Enlaces de abstracción. Un MOP tiene enlaces de abstracción a aquellos MOPs que son abstracciones suyas.

2.- Enlaces de escena. Un MOP representando sucesos, tiene un enlace de escena a los MOPs que representan sus subsucesos (otras escenas).

3.- Enlaces a ejemplos. Apuntan a casos particulares de un MOP. En algunos sistemas, serían los enlaces que conectan un MOP con los casos particulares de los cuales se derivó el MOP. En otros sistemas, estos enlaces apuntarían a ejemplos prototípicos.

4.- Enlaces de índice son los que apuntan a especializaciones del MOP. Un enlace índice está etiquetado con un par atributo-valor. Un punto clave de estos índices, es que un índice entre un MOP "padre" y un MOP "hijo" está basado en atributos y valores que no son normas del MOP "padre", ya que las normas de un MOP son verdad para todas las especializaciones del MOP. Si un MOP "hijo" se indexa por algún par atributo-valor, entonces ese par automáticamente se convierte en una norma de ese MOP. La única forma de que un suceso llegue a alcanzar ese MOP es que tenga ese par atributo-valor. Por tanto, cualquier suceso bajo ese MOP tendrá esa característica.

5.- Enlaces de fallo son los que conectan un MOP con casos particulares de éste, sucesos reales que no fueron lo que el MOP predecía. No quiere decirse que esos MOPs apuntados por ligaduras de fallo sean soluciones erróneas. Simplemente significa que no se obtuvo lo que se esperaba.

Como ya se dijo, la memoria está formada por una red de MOPs. Los enlaces entre los MOPS son los descritos en los párrafos anteriores. Así pues, podemos mirar esa red desde diversos puntos de vista, según sea el enlace que utilizamos para recorrer los MOPS.

Si miramos la memoria a través de los enlaces de abstracción, vemos una red de MOPS que va desde los casos más específicos y particulares en la parte inferior, hasta el conocimiento más general y abstracto en la parte superior. Esto se denomina jerarquía de abstracción. Las normas de cada MOP son heredadas por los MOPs que tiene debajo.

Si miramos la memoria siguiendo los enlaces de escena, vemos una red denominada jerarquía de empaquetado. En la parte de arriba están los MOPs que representan sucesos muy complejos. Estos sucesos están descompuestos en subMOPs que a su vez pueden descomponerse de nuevo hasta que lleguemos a algún conjunto de acciones primitivas no descomponibles, tales como la dependencia conceptual (Schank, 1975).

Una red discriminante (Charniack *et al.*, 1987) es lo que se observa si se mira la memoria a través de los enlaces de índice. Los MOPs están enlazados por secuencias de predicados y valores de predicados que subdividen el conjunto de subMOPs descendientes de un MOP en subconjuntos más manejables.

En nuestro sistema utilizamos dos tipos de enlaces. Los hemos denominado S-LINK y R-LINK. Los S-LINK actúan como enlaces de índice y abtracción (en la implementación no se han diferenciado). Los R-LINK serían una especie de enlaces de escena. Como

veremos en el capítulo cuarto, son los que van a indicar la secuencialidad de la solución de un problema.

2.3.3 Aprendizaje basado en casos y organización de memoria

Hay tres tipos de cambios principales que pueden ocurrir -con su consecuente aprendizaje- en una biblioteca de casos basada en MOPs: Añadir nuevos casos, nuevas abstracciones o nuevos índices.

Los nuevos casos se añaden durante el uso normal de la memoria de MOPs en la resolución de problemas.

Las nuevas abstracciones se forman cuando se descubren varios casos que comparten un conjunto común de características. Las características comunes se usan para crear las normas de los nuevos MOPs, y las características no compartidas se usan como índices a los MOPs originales. Se trata por tanto de una generalización basada en similaridad ("Similarity-Based Generalization", SBG). Este mecanismo de generalización no es exclusivo del CBR, se utiliza en otros métodos de razonamiento.

Los procedimientos para detectar si hay características compartidas, decidir si merece la pena crear una abstracción y elegir los índices, varían de un sistema a otro. Normalmente se realiza una generalización si hay bastantes casos con suficientes características en común, donde "bastante" o "suficiente" se definen por medio de umbrales. Formar nuevas abstracciones simplemente sobre la base de características compartidas no es una buena técnica, ya que por una serie de coincidencias pueden abstraerse características que no tienen darse necesariamente. Así por ejemplo, en el sistema IPP que trata sobre acciones terroristas, podría abstraerse que algunos terroristas siempre producen dos víctimas en sus atentados.

En CYRUS (Kolodner, 1984) las generalizaciones se forman por etapas. Cuando un nuevo suceso se indexa en el mismo lugar de la memoria (bajo el mismo MOP) que un suceso previo, se forma una abstracción con todas las características que los sucesos tienen en común y que no están en el MOP que tienen situado encima. Las normas de esta abstracción se marcan como potenciales porque hay muy poca evidencia de su existencia. Cuando se añadan mas sucesos a ese MOP, las normas podrán ser fijadas. Algunas características potenciales desaparecerán -ya que no son compartidas por los nuevos sucesos- y otras que no eran potenciales se añadirán porque están presentes en la mayoría de los últimos casos introducidos.

Otra solución para evitar los problemas de formar normas que luego no son generales es lo que se conoce como generalización basada en explicaciones ("Explanation-Based Generalization", EBG) (Mitchell *et al.*, 1986; DeJong & Mooney, 1986). La idea es que una abstracción se puede hacer sólo cuando se puede inferir, a partir de un conocimiento causal

previo, una razón plausible para su existencia. Se puede formar una abstracción partiendo de un solo ejemplo, siempre que el sistema pueda proporcionar suficiente causalidad para explicar las características básicas del caso. Sólo las características relevantes para la causalidad se guardan en la abstracción.

El problema con la EBG es que puede acabar haciendo un montón de trabajo para crear una abstracción de un suceso que ocurre una sola vez. Parece más razonable utilizar una técnica mixta: Cuando llega un suceso que no es similar a ningún otro se le coloca en el lugar adecuado de la memoria, pero cuando llega un suceso similar entonces se aplica EBG para crear una abstracción.

Varios sistemas CBR utilizan un tipo de EBG llamado aprendizaje conducido por fallo ("Failure-Driven Learning"). Por ejemplo, CHEF (Hammond, 1989) aprende no sólo guardando soluciones, sino formando explicaciones generales de por qué algunas soluciones no funcionan. En un sistema de CBR con aprendizaje guiado por fallo, un informe del fallo llega desde el mundo real (bien introducido por el usuario o bien generado automáticamente por el sistema al intentar ejecutar la solución). El sistema repara el caso, almacena la reparación y reorganiza la biblioteca de casos para que la reparación pueda ser encontrada en situaciones futuras similares. Esto último es importante y necesario. No es suficiente quitar ese caso de la memoria, porque entonces también se debería quitar el caso a partir del cual se generó. Sin embargo, ese caso original es correcto y sigue siendo válido en diversas situaciones. Por el contrario, lo que se pretende al guardar el caso como erróneo, es indicar al sistema cuándo no debe utilizar ese caso. Así, el sistema cambia su forma de indexar tales casos, teniendo en cuenta algunas características que no fueron tenidas en cuenta previamente.

Nuestro sistema realiza las generalizaciones de forma similar a CYRUS. Se diferencia en que las normas no se marcan como potenciales, sino que se lleva un contador. Este contador se incrementa cada vez que se accede a esa generalización con éxito, en caso contrario se decrementa. Pasado cierto umbral inferior esa información deja de estar accesible por el índice correspondiente. De esta forma el sistema mantiene bajo control el crecimiento de la memoria del sistema, al mismo tiempo que se simula el fenómeno del olvido de la memoria humana.

Una parte importante del aprendizaje consiste esencialmente en mover información que ya conocemos a los lugares adecuados. Por ejemplo, en el CHEF, ya se sabe que la carne suelta agua al cocerse y que algunos vegetales se ponen fofos si se cuecen con agua. Sin embargo no se sabe cómo utilizar esa información, de tal forma que al generar una receta produce un fallo. Una vez detectado y reparado el fallo, CHEF ya se sabe cómo utilizar esta información para no producir más fallos.

Los dos procesos de generalización que hemos mencionado -SBG y EBG-, fallan en el sentido de que dejarán sin construir algunas abstracciones potenciales. Esto se debe a que sólo intentamos formar abstracciones con los casos que están en el mismo lugar de la memoria en que hemos buscado el caso actual. Pero puede haber casos en otra parte de la memoria que comparten características con el caso actual y que no se encuentran porque las características involucradas no se están utilizando como índices.

La elección de índices es un problema difícil. Cuando se forma una abstracción, cada cosa que es compartida por los casos a partir de los cuales se generó la abstracción, se pone como norma del nuevo MOP. Todas aquellas características que no se comparten se utilizan para indexar los viejos MOPs bajo el nuevo. Sin embargo, ha de tenerse cuidado de no intentar buscar características extremas, ya que el propósito de la organización de la memoria es almacenar información disponible para usos futuros. Un índice tiene que servir para diferenciar los casos pero no debe ser único (pueden existir otros sucesos a los que les correspondiera el mismo índice).

Las características mas útiles para ser usadas en la indexación son aquellas compartidas por muchos casos particulares presentes en la memoria en general, pero compartidas sólo por unos pocos de los casos que están debajo del MOP que se está creando.

2.3.4 Adaptación de casos

La última tarea de los sistemas CBR es adaptar la solución almacenada en uno de los casos recuperados de la memoria según las necesidades de la entrada actual. Cuando el sistema recibe la situación de entrada, busca la que mejor empareja de toda la memoria. Pero, normalmente, el caso encontrado no emparejará perfectamente: habrá diferencias entre el problema que representa el caso recuperado y el problema actual. Estas diferencias se tienen que tener en cuenta.

Así pues, el proceso de adaptación busca las diferencias existentes entre el caso recuperado y la entrada, y a continuación, aplica reglas que tienen en cuenta estas diferencias.

Las reglas de adaptación son esencialmente mini-solucionadores de problemas. En un dominio de planificación -como sería el de la PA-, las reglas necesitan poder anotar las precondiciones a los pasos que necesitan ser satisfechas y encontrar planes para alcanzar dichas precondiciones.

Como discute Hammond (1989) la ventaja del CBR es que las reglas de adaptación pueden ser mucho mas simples que las requeridas en un sistema basdo en reglas, siempre que la biblioteca de casos esté razonablemente llena. En muchos dominios reales es muy difícil, si no imposible, crear un conjunto completo de reglas y además aplicar grandes

cantidades de reglas es muy ineficiente. Un sistema CBR puede funcionar con un conjunto pequeño de reglas si la biblioteca de casos es extensa. Además las reglas tienen que ser mucho menos potentes ya que sólo se usan para poner parches a las soluciones, no para crear una solución de la nada.

Tipos de adaptación

Existen dos clases de adaptaciones muy diferentes descritas en la literatura del CBR. La adaptación estructural, en la que las reglas de adaptación se aplican directamente a la solución almacenada en un caso, y adaptación derivacional, donde las reglas que generaron la solución original se reejecutan para generar la nueva solución. En este último tipo, de lo que se trata es de almacenar -no sólo una solución con un caso- sino la secuencia de planificación que construyó dicha solución. Cuando se recupera un caso, el sistema comprueba si las diferencias entre la vieja situación y el caso de entrada afectan a alguna de las decisiones subyacentes a la antigua solución. Si es así, dichas decisiones son reevaluadas utilizando los valores existentes en la entrada. Es decir, la solución se adapta no cambiándola directamente sino reejecutando partes del proceso original de solución.

La adaptación derivacional tiene varias ventajas. Primero, se necesitan muy pocas reglas *ad hoc*. No se necesitan reglas para eliminar pasos inútiles, ya que estos pasos nunca aparecen cuando los planes se reejecutan en nuevas circunstancias.

Segundo, la adaptación derivacional puede utilizarse para adaptar conocimiento de resolución de problemas de otros dominios, en lugar de estar restringida a soluciones dentro del dominio. MEDIATOR (Simpson, 1985) es un caso de razonamiento analógico, cuyo funcionamiento sólo es posible con adaptación derivacional.

La adaptación derivacional no reemplaza la estructural. En cualquier CBR real estarían presentes ambos mecanismos, porque la adaptación derivacional depende de la presencia de estructuras de planificación para las soluciones almacenadas y no todas las soluciones las tienen.

Un CBR completo tendrá reglas de adaptación estructural para arreglar casos con soluciones no analizadas y mecanismos derivativos para casos bien entendidos por el sistema. Las soluciones generadas por el propio sistema serían las primeras candidatas para adaptación derivacional.

Técnicas de adaptación

En esta sección daremos una panorámica de diferentes técnicas de adaptación que han sido utilizadas en distintos sistemas de razonamiento basado en casos. Estudiaremos técnicas de adaptación estructural y derivacional. Empezaremos por la adaptación "nula"; estrictamente hablando, no es ni estructural ni derivacional.

Adaptación "nula"

La primera técnica, y ciertamente la más simple, es no hacer nada y simplemente aplicar -cualquiera que sea- la solución recuperada, a la nueva situación.

La adaptación "nula" aparece en aquellas tareas donde el razonamiento necesario para una aplicación puede ser muy complejo, y la solución - en si misma- muy simple. Por ejemplo, pueden considerase muchos factores cuando se evaluan recursos a multas de tráfico, pero la respuesta es aceptarlos o rechazarlos. Si el caso que mejor empareja con el problema de entrada rechaza el recurso, entonces el sistema lo rechaza; si el caso que mejor empareja, acepta dicho recurso, entonces se acepta. En los dos casos la solución del caso recuperado de la memoria se aplica directamente. Un programa que diagnostique fallos de equipos basándose en los fallos de sus componentes, puede trabajar de forma similar.

Con este tipo de soluciones tan simples (aceptar, rechazar, engranaje- 25, etc.) no sorprende que no haya mucho que pueda ser adaptado. Sin embargo, hay que tener en cuenta dos consideraciones: Primero, si todo lo que nosotros queremos es una respuesta simple, existen otras técnicas, como los métodos de clasificación estadísticos, que pueden trabajar muy bien -a menudo mejor que la gente- y que son más convincentes. Segundo, es raro que queramos este tipo de soluciones tan simples. El que evalúa la petición de un recurso no expone simplemente el resultado final, sino que debe explicar toda la cadena de razonamiento a la persona que lo solicitó.

Soluciones parametrizadas

Probablemente, la técnica de adaptación mejor comprendida es una técnica estructural de soluciones parametrizadas. La idea es que cuando se recupera un caso para una entrada concreta, las descripciones del nuevo problema y del viejo se comparan con respecto a unos parámetros especificados. Las diferencias se usan para modificar los parámetros de la solución en las direcciones apropiadas. Cada parámetro del problema se asocia con uno o mas parámetros de la solución.

El uso de soluciones parametrizadas no implica que haya una fórmula simple para conseguir, a partir de algún conjunto de parámetros del problema, una solución.

Una limitación de esta técnica es que las soluciones parametrizadas son válidas para modificar una solución pero no para crear una nueva solución *ab initio*. Son una técnica de interpolación. Es una forma simple pero potente de aumentar una biblioteca de casos pero no sustituye a un buen conjunto de éstos.

Los sistemas HYPO (Rissland & Ashley, 1986; Ashley & Rissland, 1988) y PERSUADER (Sycara, 1987) utilizan esta técnica.

Abstracción y reespecialización

Es una técnica de adaptación estructural muy general que puede ser utilizada tanto a nivel básico, para conseguir adaptaciones simples, como a nivel complejo, para obtener nuevas y hasta creativas soluciones.

La idea es la siguiente: si un trozo de la solución recuperada no es aplicable para el caso actual, se buscan abstracciones de ese trozo de solución que no tengan la misma dificultad. A continuación, se reespecializa, es decir, se intentan aplicar a la situación actual otras especializaciones de la abstracción. Se obtienen de esta forma hermanos o parientes colaterales del concepto original.

Este tipo de adaptación la encontramos en PLEXUS (Alterman, 1986).

Adaptación basada en crítica

La técnica de resolver problemas utilizando críticas para depurar soluciones casi correctas, fué primero propuesta por Sussman (1975) y desarrollada más recientemente por Simmons (1988). Aunque las soluciones básicas en estos sistemas se crean simplemente combinando reglas, más que recuperándolas de memoria, las críticas de la solución son muy similares en espíritu, y a veces en contenido, a las críticas usadas en CBR.

Una crítica busca alguna combinación de características que puede causar un problema en un plan. Existen estrategias de reparación asociadas con diferentes situaciones problemáticas. En el trabajo de Sussman, un plan -que debe alcanzar varios objetivos simultaneamente- se deriva poniendo juntos los planes que podrían alcanzar cada objetivo independientemente. Las críticas pueden entonces comprobar si algún plan interfiere con otro o si hay planes redundantes.

CHEF por ejemplo, obtiene una solución básica tomándola de una receta anterior y sustituye los nuevos ingredientes deseados por los viejos. A partir de entonces, unas críticas sobre los ingredientes chequean pasos innecesarios -como cortar ingredientes que ya son pequeños- o detectan la carencia de pasos como deshuesar aquellas carnes que deben cortarse. Más tarde, cuando se repara una receta que no ha funcionado, las críticas de reparación separan los pasos que interfieren unos con otros.

Lo mismo que se dijo en las soluciones parametrizadas también se puede decir aquí. Las críticas no son capaces de generar soluciones completas por sí solas. Están limitadas al tipo de parches que pueden hacer en las soluciones. Hacen cambios locales en vez de reorganizaciones globales. La forma básica de la solución final todavía refleja la solución original de la que derivó.

Reparticularización

Las técnicas descritas hasta ahora estaban comprendidas dentro de las que se han llamado métodos de adaptación estructural. Operan directamente sobre las viejas soluciones para producir las nuevas. La reparticularización ("Reinstantiation") es un método derivacional. No opera sobre la solución original sino sobre el método que fué usado para generar dicha solución. Reparticularización significa reemplazar un paso en el plan que generó la solución y reejecutarlo en el contexto de la situación actual. La potencia de la reparticularización está limitada por la potencia de planificación del sistema, puesto que reparticularizar un plan es planificar.

MEDIATOR (Simpson, 1985) utiliza esta técnica para generar soluciones a disputas entre dos partes. Por ejemplo, intenta resolver la disputa entre Egipto e Israel sobre el control del Sinaí, utilizando la solución que se aplicó cuando el conflicto entre EEUU y Panamá sobre el control del canal de Panamá. El plan que se utilizó fue "divide en partes diferentes", separando el control del canal en dos partes: control militar y control político. En la situación actual, una solución similar es dar el control militar a Israel, para su seguridad nacional, y el control político a Egipto para su integridad nacional.

Otro sistema que utiliza esta técnica dentro de la programación automática es APU (Bhansali, 1991).

2.3.5 Explicación y reparación

Cuando un CBR falla tiene que explicar su fallo y repararlo. En algunos dominios, primero se dá la explicación y la reparación se basa en la explicación. En otros dominios primero se hace la reparación, y sólo entonces se puede dar una explicación del fallo.

Hay dos tipos de fallos cuando se generan planes u otros diseños:

- No se alcanzan algunos objetivos especificados en la entrada. Por ejemplo un plan no hace lo que se supone debería hacer.

- Se violan algunos objetivos implícitos, no especificados en la entrada. Por ejemplo se consigue la meta especificada pero su coste es demasiado elevado.

Cuando un objetivo implícito es violado lo primero que debe hacerse es ponerlo de forma explícita para la situación de entrada actual. De esa forma será visible cuando se hagan futuras recuperaciones.

Explicación

La tarea del proceso de explicación es generar una explicación específica del dominio de por qué falló la solución propuesta.

La naturaleza del proceso de explicación, actualmente, es una materia en estudio. CHEF y COACH usan reglas para generar sus explicaciones, pero la complejidad del proceso de explicación sugiere que es preferible utilizar también aquí CBR en vez de RBR. Otros sistemas como SWALE, adaptan las explicaciones (Schank, 1986). Este enfoque es necesario porque existen demasiadas posibles explicaciones para eventos anómalos, y debemos restringir la búsqueda a clases de explicaciones conocidas.

Reparación

La reparación es similar a la adaptación porque consiste en modificar una solución para ajustarla a una situación. La diferencia está en que la adaptación comienza con una vieja solución y un caso nuevo y adapta la solución a la nueva situación. La reparación parte de una solución, un informe de fallo, y quizá una explicación, y modifica la solución para eliminar el fallo.

En dominios donde la explicación precede a la reparación, la explicación del fallo suele proporcionar pistas sobre las reparaciones necesarias. Por ejemplo en CHEF, las explicaciones están enlazadas por enlaces de abstracción a los TOPs ("Thematic Organization Packages") y unidos a los TOPs hay heurísticas generales para reparar el problema (independientes de dominios particulares). Este modelo de reparación no es aplicable a dominios en los que no es posible dar explicaciones hasta que una solución correcta ha sido encontrada (por ejemplo, diagnóstico de averías).

Cuando la única información disponible es que la solución propuesta no funciona, una estrategia de reparación consiste en añadir cualquier información complementaria que aparezca en el informe de fallo y repetir la búsqueda en la biblioteca de casos con toda esa información. Si el caso recuperado es distinto del recuperado la primera vez entonces el nuevo caso recuperado debe ser adaptado como nueva solución. Si el caso recuperado fuera el mismo que la primera vez, otra estrategia consistiría en elegir el segundo mejor caso seleccionado en vez del mejor caso posible.

Siempre que hay soluciones que fallan y se reparan, por el medio que sea, una cosa importante es guardar un enlace entre la solución que no funcionó y la que finalmente lo hizo. Este enlace de fallo permitirá que cuando en otra situación similar se produzca un fallo, el sistema pueda explorar todos los fallos asociados a ese caso e intentar generalizar lo que es común -si es que lo hay- entre esas ocasiones en las que el caso no ha funcionado. Tanto las técnicas de generalización basadas en similitud como las basadas en explicaciones serían aplicables aquí. El objetivo es encontrar alguna caracterización de las situaciones en que falla una solución. Con esta caracterización el sistema sería capaz de solucionar por si sólo esa clase de fallos y evitarlos en el futuro. Al menos, el sistema será capaz de saber cuándo está en situaciones en las que ha tenido dificultades previamente.

En el estado actual, nuestro prototipo no cuenta con mecanismos de explicación y reparación

2.4 ALGUNOS SISTEMAS BASADOS EN CASOS

A continuación, vamos a describir algunos sistemas de CBR. De los aquí descritos, el que más puede interesarnos es el CHEF ya que trabaja - como la programación- en un dominio de diseño.

IPP

IPP (Integrated Partial Parser) (Lebowitz, 1983) trata de modelar la forma en que la gente lee y entiende determinados tipos de historias. Desde cómo se procesa un texto, hasta su integración en la memoria y posterior generalización.

Aunque el IPP se centra principalmente en el proceso de generalización, también está orientado a crear expectativas, que puedan ser utilizadas por un *parser*, para resolver situaciones ambiguas y traer a memoria información implícita en el contexto. Con este sistema se trata de dar respuesta a preguntas como: ¿Por qué generalizar es una parte integrante del proceso de comprensión?, ¿Cómo reconocer cuándo es necesario generalizar? y ¿Cómo evaluar el grado de acierto de las generalizaciones?. También se pretendía encontrar un esquema de representación que facilitara este proceso de generalización.

El modelo presentado en IPP organiza la memoria en términos de generalizaciones. "Entender" una nueva información en la entrada equivale a acceder a informaciones previas que sean similares a la nueva.

Debemos precisar a qué nos referimos cuando decimos que el sistema "entiende". El dominio de IPP son noticias sobre actos terroristas. Uno de los ejemplos que utiliza Lebowitz se refiere a dos noticias sobre ataques terroristas llevadas a cabo por palestinos en el Medio Oriente. Estas dos noticias permiten al sistema crear la generalización de que en el Medio Oriente los ataques terroristas suelen realizarlos palestinos. Al recibir una tercera noticia sobre la colocación de una bomba en un hospital de Hebron, en la que no se menciona la identidad de los autores del atentado, la generalización obtenida anteriormente permite al sistema inferir que los autores pueden ser palestinos.

Cuando IPP sólo tienen en su memoria tres o cuatro noticias de terrorismo -tres o cuatro casos-, como en el ejemplo anterior, debemos ser conscientes de que, en este contexto, "entender" tiene un significado ni remotamente parecido al que tiene para nosotros. El sistema se limita a extraer la parte común de tres o cuatro estructuras de datos -casos- y a utilizarla para construir un índice en la memoria, pero los símbolos que componen estas estructuras carecen de toda referencia en otras partes del sistema (aunque

IPP tiene alguna información asociada a cada uno de los posibles datos concretos, como por ejemplo, que Italia es un país del sur de Europa Occidental, que su sistema político es una democracia, etc).

El problema con IPP es, no obstante, lo limitado de su dominio y que está basado en unas estructuras (S-MOP) elaboradas externamente al sistema. Los S-MOP ("Simple - Memory Organization Packages"), describen los conceptos abstractos necesarios para organizar la información en su dominio -por ejemplo: ataque, extorsión, opresión de minorías, etc.- donde se establecen las relaciones causales entre los componentes de cada esquema. Sin embargo, no referencian ninguna otra estructura previa, ya que estas estructuras forman la raíz de la memoria, es decir, su parte superior.

Las generalizaciones que, como la citada más arriba de los palestinos, hacen referencia a actos concretos, que pueden interpretarse en el contexto de los S-MOP, reciben el nombre de "unidades de acción" (un concepto similar al de *script*), también llamados spec-MOP, y son, junto los S-MOP, las estructuras que organizan la memoria del sistema.

Los S-MOP son patrones o plantillas en los que se capturan los esquemas causales que subyacen a situaciones aparentemente diferentes. Por ejemplo, el concepto de extorsión, que implica una amenaza y algo que debe hacer la víctima, o alguién relacionado con ella, para evitar la amenaza, puede estar subyacente en historias de secuestros, raptos, ataques terroristas, etc. La hipótesis de partida es que la información de los S-MOP es la que normalmente se deja implícita cuando se describe un hecho concreto.

El esquema de organización de la memoria en IPP es por tanto el siguiente: inicialmente la raíz está formada por un conjunto de S-MOP (que describen los conceptos abstractos del dominio) y bajo ésta se va indexando una red de spec-MOP, creada a partir de los episodios concretos. Los episodios concretos están indexados bajo los spec-MOP por sus diferencias. La red de spec-MOP es dinámica en el sentido de que nuevos spec-MOP son creados a partir de generalizaciones de episodios concretos.

Las generalizaciones que el sistema crea a partir de noticias sobre actos terroristas son del tipo: "Las acciones terroristas en Italia son sobre hombres de negocios y no suelen producir la muerte", "en el país vasco los ataques terroristas suelen producir víctimas mortales", "tomas de edificios y condiciones para liberar rehenes son frecuentes en América Hispana", etc.

Estas generalizaciones no hacen explícita ninguna relación de causa- efecto. Por tanto, se crea un conflicto en la memoria de estos sistemas si, en determinado momento, se introduce una relación causa- efecto que cambia la situación, como, por ejemplo, que el narcotráfico haya cambiado las características del terrorismo en la América Hispana.

La principal limitación de IPP es, por tanto, que no se enfrenta al problema de cómo se crean estos patrones iniciales en la memoria, sólo trata de reconocer que un hecho concreto puede ser explicado por ellos y crea esquemas (spec-MOP) más específicos a partir de ellos.

CYRUS

CYRUS (Kolodner, 1983) introduce explícitamente la idea de que, para lograr las características de la memoria humana, la base de conocimientos debe organizarse en categorías conceptuales, categorías basadas en semejanza de significado. En líneas generales, el concepto de generalización propuesto es el mismo que en IPP, aunque añade la idea del control sobre los índices, debido al mayor número de detalles del dominio elegido.

Las diferencias entre CYRUS e IPP son, en buena medida, el diferente tipo de información que ambos sistemas contienen. Los esquemas de CYRUS hacen referencia a episodios personales (de los secretarios de estado Cyrus Vance y Edmund Muskie) y tienen un mayor número de detalles, mientras que los de IPP (actos terroristas), por el contrario, tienen menos riqueza de detalles, pero obedecen a situaciones abstractas bien definidas y para las que la gente tiene esquemas predeterminados, por ejemplo, extorsión, opresión de las minorías, etc.

La estructura básica se llama E-MOP ("Episodic - Memory Organization Packages"). Estos E-MOPs se consideran como "categorías conceptuales" y son los que organizan la memoria. Estas estructuras reflejan el dominio del sistema, en este caso las actividades de un secretario de estado. Por ejemplo, hay E-MOPs referidos a reuniones diplomáticas, viajes, tratados internacionales, etc. Los E-MOPs tienen dos componentes: los índices a otros E-MOPs que forman una estructura tipo árbol, y el contenido, dividido a su vez en normativo (personas que intervienen, lugares, etc.) y su contexto (relaciones con otros E-MOPs). De igual forma que en IPP, los MOPs de CYRUS describen situaciones complejas previamente determinadas. Los *roles* (o *slots*) están muy orientados al dominio: temas que se tratan en las reuniones diplomáticas, participantes en las mismas, etc.

La estructura de la memoria es básicamente la misma que en IPP: el conjunto inicial de E-MOPs que describen el dominio (en forma más detallada que en IPP) y luego los E-MOPs formados por el sistema cuando indexa nuevos episodios. Los episodios se indexan por todas sus diferencias con los E-MOPs en memoria y cuando hay más de un episodio indexado por la misma característica bajo un E-MOP se forma uno nuevo, especialización del anterior, extrayendo para ello las características comunes a los episodios y reindexando dichos episodios bajo la nueva estructura.

Una diferencia con IPP es que, al ser la información más rica en detalles, existe un mayor número de índices que hay que mantener bajo control, mediante una selección más rigurosa de los mismos.

Los mecanismos de selección de índices en CYRUS están guiados estadísticamente. La idea de fondo es tratar de agrupar episodios similares y extraer de ellos lo que podríamos llamar un prototipo o episodio característico, para después organizarlos en una jerarquía de especializaciones a partir de una categoría inicial, como por ejemplo el E-MOP "*reuniones diplomáticas*".

Así pues, las únicas categorías conceptuales que hay son las iniciales. Todo lo demás consiste en organizar episodios por sus similitudes bajo estas características iniciales.

CYRUS implementa explícitamente dos características a tener en cuenta:

La primera es el caracter reconstructivo de la memoria. Una memoria reconstructiva es una memoria de la que recuperamos la información precisando más y más las características del ítem que buscamos. Una memoria de este tipo requiere estar organizada en torno a conceptos distintos a los IS-A , INSTANCE-OF, etc. Si la memoria humana fuera una jerarquía de este tipo no se producirían olvidos, y si un concepto tiene cientos de particularizaciones habría que recorrerlas todas para encontrar una concreta, en contra de lo que se observa que hacen los expertos humanos. La conclusión es que debe haber muchos más tipos de arcos o índices, tantos como diferencias entre cada particularización y el concepto bajo el cual de indexa, de forma que, estrechando pregresivamente las características de un ítem, podamos llegar directamente hasta él.

La segunda característica es consecuencia de mantener asociados en la misma base de datos las generalizaciones y los ítems a partir de los cuales estas generalizaciones han sido creadas. Esto hace que el proceso de recuperar los ítems sea el mismo que el de recuperar las generalizaciones.

CHEF

CHEF (Hammond, 1989) es un planificador basado en casos, cuyo dominio son las recetas de cocina. CHEF crea las nuevas recetas a partir de otras ya conocidas, como respuesta a unos requisitos de platos con unos ingredientes y sabores particulares. CHEF tiene que construir planes que satisfagan un número dado de metas simultáneamente.

Cuando se pide una nueva receta a CHEF con unas características determinadas, lo primero que hace es buscar un plan (una de las recetas que ya conoce) que satisfaga el mayor número posible de las características o metas que se solicitan. Una vez ha encontrado ese plan, lo modifica para que cumpla todas aquellas características que aún no satisface.

CHEF altera estas recetas con reglas de modificación y un conjunto de "objetos críticos". Estas reglas son específicas de la clase de plato de que se trata (frito, soufflé, etc.) y de la característica a modificar. Los "objetos críticos", son los que cambian los tiempos de cocción de los ingredientes, teniendo en cuenta los cambios que ha sufrido la receta inicial.

Una vez hechas estas modificaciones, se ha conseguido una receta que debería verificar todas las condiciones que se le han impuesto.

Sin embargo, puede darse el caso de que el resultado no sea correcto. En algunos sistemas de CBR es el usuario el que indica que la respuesta es errónea. En el caso del CHEF, es el propio sistema (a través de un simulador de la ejecución del plan) quien detecta el error.

Para CHEF, un error en un plan significa dos cosas: la necesidad de reparar la receta, y de "reparar" la comprensión que tiene del mundo que le llevó a cometer el error.

Para estas dos tareas, CHEF necesita comprender exáctamente el motivo por el cual se dió el fallo. No es suficiente que pueda describir en qué consiste el fallo, sino que debe ser capaz de explicar por qué ese fallo concreto ha ocurrido.

CHEF explica el fallo a través de una descripción causal de por qué sucedió. La explicación describe el fallo, el paso que lo causó y las condiciones que deben cumplirse para que éste se presente.

Esta explicación tiene dos funciones. Primero describe el problema de la planificación en un vocabulario causal de tipo general que puede usarse para acceder a alguna estrategia general de reparación. Segundo, esta explicación señala qué características han interaccionado para causar ese fallo, y por tanto en qué características se debería estar atento en un futuro.

Las estrategias de resolución están indexadas bajo TOPs (Schank, 1982). Los TOPs son similares a los MOPs, e indexan un tipo particular de errores de planificación que son capaces de reparar. CHEF utiliza la explicación causal para buscar el TOP que tiene las estrategias que reparará el error.

Una vez el plan se ha corregido, además de indexarlo como solución errónea (para no volverlo a repetir), CHEF también cambia su comprensión del mundo, de forma que sea capaz, en otra ocasión similar, de evitar ese fallo. Su comprensión del mundo la cambia creando nuevas abstracciones: una abstracción que describe las caracterísicas que condujeron al fallo, bajo la cual pone la receta reparada para poderla utilizar en la realización de futuras recetas, y otra abstracción para el resto de las situaciones, bajo la que se indexan todas las recetas que no presenten esas características que conducen al fallo.

CHEF aprende de los planes que genera así como de sus fallos. Los nuevos planes que obtiene los indexa tanto por las metas que consiguen como por los problemas que evitan. Al mismo tiempo, al indexar los fallos como tales con las características que los provocaron, es capaz de anticiparse a esos problemas cuando se le soliciten de nuevo recetas con esas características.

Como ya se mencionó, CHEF es uno de los sistemas más próximos al nuestro. La parte más similar es la relativa a las reglas de adaptación. En el CHEF, estas reglas o precondiciones indican pasos que hay que añadir o quitar a la receta según determinadas circunstancias que podían no presentarse en la receta recuperada de la memoria, pero que en el caso actual si se presentan. Por ejemplo, una precondición de CHEF es cortar aquellos ingradientes que sean grandes, o deshuesar las carnes con hueso. En nuestro sistema, estas precondiciones nos llevarán a añadir sentencias como pueden ser las de inicialización de variables.

2.5 SISTEMAS DE SINTESIS DE PROGRAMAS BASADOS EN ANALOGIAS

Hasta ahora hemos descrito algunos sistemas basados en el conocimiento, que utilizan un razonamiento basado en casos, y que funcionan en dominios muy variados. En este apartado, describiremos algunos sistemas basados en analogías cuyo dominio es la síntesis de programas.

Aprendiz de Programador

El Aprendiz de Programador (Rich, 1981; Rich & Waters, 1988) es un sistema interactivo para ayudar al programador en la tarea de la programación. Lo que se pretende es que el programador realice las partes más difíciles del diseño y de la implementación, mientras que el aprendiz realiza una tarea de asistente en la documentación, verificación, depuración y modificación de los programas.

Para cooperar de esta manera con el programador, el aprendiz debe ser capaz de "comprender" lo que se está haciendo. Desde el punto de vista de la Inteligencia Artificial el trabajo principal desarrollado con el Aprendiz de Programación ha sido el diseño de una representación para los programas, los *planes*, y para el conocimiento de programación que sirve como base de esa comprensión. El desarrollo de los planes y la comprensión sobre ellos, es la actividad central del Aprendiz de Programador.

El plan para un programa representa el programa como una red de operaciones conectadas por enlaces explícitos que representan el flujo de datos y el flujo de control. El plan no es sólo un grafo de operaciones primitivas, sino que es una jerarquía de segmentos, donde cada segmento corresponde a una unidad de comportamiento y tiene una especifi

cación de entrada/salida que describe las características de su comportamiento. La segmentación es importamte porque rompe el plan en trozos que pueden ser comprendidos independientemente unos de otros.

El comportamiento de un segmento está relacionado con el comportamiento de sus subsegmentos. Esta relación se establece por enlaces de dependencia explícitos que indican las relaciones entre los subobjetivos y precondiciones de la especificación entrada/salida del segmento con los de los subsegmentos.

El conocimiento de programación, en general, también se representa por planes y descripciones de estructuras. Este conocimiento se almacena en una base de datos de algoritmos comunes e implementaciones de estructuras de datos llamada "bliblioteca de planes (*plan library*).

SPECIFIER

SPECIFIER (Miriyala & Harandi, 1991) es un sistema interactivo que guia al usuario para derivar especificaciones formales de tipos de datos y de programas a partir de sus descripciones informales.

El proceso de la especificación comienza con una definición informal del problema expresado en un subconjunto del lenguaje natural. Un preprocesador analiza esta especificación informal y extrae los conceptos importantes que allí se nombran.

Los conceptos representan tipos de datos u operaciones sobre datos. Asociado a cada concepto existe una estructura tipo *frame* que representa la información semántica del concepto. El preprocesador toma esas *frames* de la base de conocimientos y llena los *slots* correspondientes utilizando la especificación informal. Reúne todas las *frames* particularizadas para obtener un análisis de las frases de la especificación informal. Este análisis se representa jerárquicamente en una estructura llamada *structure tree* (árbol de estructura).

Este árbol es la salida del preprocesador y es lo que utiliza el razonador del sistema.

Primero, el razonador basado en esquemas intenta encontrar un esquema en la base de conocimientos que sea aplicable al problema que se está tratando. Un esquema es de la definición abstracta de un operador. El sistema particulariza ese esquema encontrado con la información que aporta el árbol de estructura para obtener la especificación formal del problema.

Si no se encuentra un esquema aplicable, el razonador analógico intenta encontrar problemas anteriores en la base de conocimientos que sean análogos al actual. Si localiza un problema análogo, aplica una correspondencia para adaptar la especificación formal del problema anterior y así obtener la especificación formal para el problema actual.

En esta situación el sistema puede llamarse a sí mismo para resolver subproblemas del problema actual. Si hay subproblemas no especificados, el sistema pide más información al usuario. Este proceso termina cuando todos los subproblemas están especificados formalmente. Esta especificación completa se pasa a un postprocesador.

El postprocesador intenta simplificar axiomas de la especificación formal y se asegura que la forma final de la especificación esté conforme con la sintaxis del lenguaje de especificación formal elegido.

APU

APU (Bhansali, 1991) es un sistema que sintetiza programas para el intérprete de órdenes de Unix a partir de especificaciones en alto nivel de problemas básicos.

Los dos bloques principales del sistema son una base de conocimientos y un generador de programas.

La base de conocimientos consta de tres partes: un diccionario de conceptos, una biblioteca de componentes reutilizables y un conjunto de reglas.

El diccionario de conceptos contiene una descripción del vocabulario dependiente del dominio (objetos, funciones y predicados) necesario para describir los problemas a alto nivel. Usando este vocabulario, el usuario puede comunicarse con el sistema al nivel de su aplicación. Los conceptos están organizados en una abstracción jerárquica con herencia de propiedades desde los niveles más altos a los más bajos. Esta jerarquía es la base del sistema para reconocer analogías entre los problemas.

El generador de programas usa la biblioteca de componentes software reutilizables para evitar la síntesis repetitiva de programas, o fragmentos de programas, que se usan frecuentemente. Dos de los tres tipos de componentes que hay son estáticas: las subrutinas y las plantillas (*templates*). La tercera, historia derivacional, (*derivation history*) la adquiere dinámicamente el sistema al resolver los problemas propuestos por el usuario. Una historia derivacional es una traza de la derivación del programa a partir de su especificación y se utiliza para acelerar la síntesis de programas análogos.

El conjunto de reglas, formando tres capas, contiene un catálogo de reglas de estrategia, de resolución de problemas y específicas del dominio, que pueden utilizarse para transformar una especificación del problema en un programa para el intérprete de órdenes.

La estructura en capas de esta base de reglas permite que el sistema se pueda configurar para diferentes dominios y otros lenguajes con pocos cambios.

El generador de programas consta de dos partes: un planificador y un razonador analógico.

El planificador, basado en un concepto de planificación jerárquica, usa la base de reglas y ciertas heurísticas para asegurarse que los fallos van a poder detectarse pronto y que se escogerán los planes eficientes se escogerán antes de los no eficientes. El planificador usa una descomposición *top-down* para generar un plan que resuelva el problema y a continuación emplea un conjunto de transformaciones simples para convertir el plan en un programa "shell".

El razonador analógico está basado en el paradigma de la analogía derivacional de Carbonell (1986), y complementa el papel del planificador. Este razonador analógico, puede recuperar automáticamente problemas de la biblioteca de historias de derivación que sean análogos al problema actual; reejecuta la traza derivacional para sintetizar programas más eficientemente que el planificador; y finalmente, almacena la traza derivacional del problema resuelto para poderla utilizar en un futuro.

Aunque lo que Bhansali defiende es un sistema que integre juntas ideas de la ingeniería del software (como reutilización de software y modelización del dominio) con las de la inteligencia artificial (como planificación jerárquica y razonamiento analógico), donde más se ha centrado su trabajo ha sido en demostrar la eficiencia de un razonador analógico. Con este razonador analógico consigue acelerar el proceso de síntesis de un programa con respecto a la síntesis a base de reglas y sin analogía.

Tres son los procesos cruciales que constituyen un razonador de este tipo que reejecuta derivaciones anteriores: (1) recuperar un caso apropiado de la biblioteca de historias de derivación, (2) adaptar esa derivación según las necesidades del caso actual, y (3) almacenar el resultado final en la biblioteca.

De estos procesos, ha implementado los dos primeros, y sugiere algunas ideas de cómo podría realizarse el tercero.

Aunque este sistema utiliza la analogía sólo para acelerar el proceso de síntesis de los programas, tiene algunas características comunes con nuestro sistema.

En primer lugar, nuestra base de conocimientos también consta de un diccionario de conceptos, de una biblioteca de casos, y de un conjunto de reglas.El diccionario de conceptos, como el de APU, describe el vocabulario dependiente del dominio, y sus conceptos están jerarquizados.

El contenido de nuestra biblioteca de componentes, difiere respecto al contenido de la biblioteca de APU. Existen tres tipos distintos de información: los casos -problemas con sus soluciones-, patrones - métodos generales de resolución- y esquemas -que, como ya veremos, son distintos de los que encontramos en APU-. Por otro lado, en APU es el diseñador del sistema el que introduce los esquemas en la base de conocimientos; en

nuestro caso, es el propio sistema quien genera los patrones y esquemas. Lo que supone una ventaja si se tiene en cuenta que la codificación de esquemas es un trabajo engorroso y para el que se requiere bastante tiempo (Miriyala & Harandi, 1991).

Nuestras reglas son parecidas a las de APU en el sentido de que tanto las unas comno las otras indican los pasos que hay que realizar para alcanzar una operación concreta. Difieren en que las reglas de nuestro sistema son más flexibles, debido a que es un sistema CBR.

Existen otros sistemas de síntesis de programas que utilizan la reejecución de planes o derivaciones anteriores como XANA (Mostow & Fisher, 1989), POPART (Wile, 1983) y KIDS (Smith, 1990), pero su atención la centran en el diseño del proceso de aplicar una regla detrás de otra, más que en la analogía.

PM

PM (Partial Metrics) (Reynols *et al.*, 1990) . No es un sistema propiamente de CBR, sino que se ocupa de lo que sería una fase previa al CBR. Es un sistema de aprendizaje de conocimientos de codificación.

Se le dan al sistema programas completamente codificados (provenientes de textos de programación básicos). El sistema, empezando por el final, intenta ir formando trozos de programas que sean lo suficientemente independiente de los demás.

Si añadir un trozo de programa a lo que ya tiene seleccionado hace que éste sea demasiado complejo, o que surjan dependencias, lo deja y toma como trozo de programa independiente el que ya tenía seleccionado. Este trozo se sustituye por un nombre, y se sigue el proceso

Para determinar la complejidad y las dependencias se usan métricas de complejidad y técnicas de ramificación y acotación (*branh and bround*), poniendo como cota el incremento de la complejidad en cada paso.

Una vez determinados los trozos, se generalizan utilizando aprendizaje simbólico (algoritmos genéricos). Esta generalización produce una descripción del trozo indicando su posición en la jerarquía de refinamientos y su funcionalidad. Esta descripción se utiliza como índice para almacenar ese trozo (caso) en una biblioteca de CBR.

En este capítulo, hemos introducido los sistemas basados en el conocimiento. Nos hemos extendido con más detenimiento en los sistemas de razonamiento basados en casos. Después de explicar la metodología de estos sistemas, hemos descrito algunos sistemas

basados en casos funcionando en dominios muy diferentes, así como algunos de los sistemas de síntesis de programas basados en analogías.

En el siguiente capítulo, describimos el modelo de memoria humana que utilizaremos en nuestro sistema y que constituye su base de conocimientos.

3. UN SISTEMA DE MEMORIA DINAMICA

3.1. INTRODUCCION

En el capítulo anterior se ha puesto de manifiesto el papel clave que juega el conocimiento del dominio en cualquier sistema de inteligencia artificial. Para que este conocimiento pueda utilizarse de forma "inteligente", no basta con integrarlo en una base de datos convencional, sino que se necesita una base de conocimientos sobre la que puedan actuar diversos métodos, que son los que "comprenden" y manejan ese conocimiento presentando un comportamiento "inteligente" (Frost, 1986).

Nuestro objetivo ha sido desarrollar un sistema que permita sintetizar programas por medio de la detección de analogías con otros programas previamente desarrollados. Nuestro trabajo se ha centrado en los métodos necesarios para inferir las soluciones a los casos que se le planteen al sistema.

Hemos tomado como modelo de memoria el propuesto por Fernández- Valmayor y Fernández Chamizo (Fernández-Valmayor, 1990; Fernández- Valmayor & Fernández Chamizo, 1992). Esta memoria contituye la base de conocimientos del sistema que presentamos. Para poder describir nuestro trabajo, primero será necesario explicar, con un mínimo de detenimiento, el modelo de memoria dinámica que desarrollan estos autores en su trabajo. Haremos constar aquellas modificaciones que han sido necesarias debido a los distintos objetivos y dominios de aplicación que se han tomado.

El objetivo de Fernández-Valmayor y Fernández Chamizo, era desarrollar un modelo de memoria que fuera un modelo cognitivo de la memoria humana y tomaban como dominio de aplicación la parte de la física correspondientea a la mecánica elemental.

Nuestro objetivo es desarrollar métodos basados en la analogía para la construcción de programas, y nos hemos limitado al dominio que correspondería a la resolución de ejercicios en una asignatura de estructuras de datos y algoritmos.

El modelo de memoria que hemos tomado, está constituído por una red discriminante que se crea de forma dinámica.

Las estructuras de datos que componen la red se crean, mediante los procesos que detallaremos más adelante, a partir de la información que recibe el sistema. Este organiza la información que recibe para formar lo que Kolodner (1983) llama "una red conceptual bloqueada". La finalidad primaria de esta red es hacer fácilmente accesible la información que el sistema necesita.

Mediante el siguiente diagrama de bloques, podemos representar nuestro sistema. El bloque denominado "MEMORIA" y los procesos necesarios para mantener esta estructura

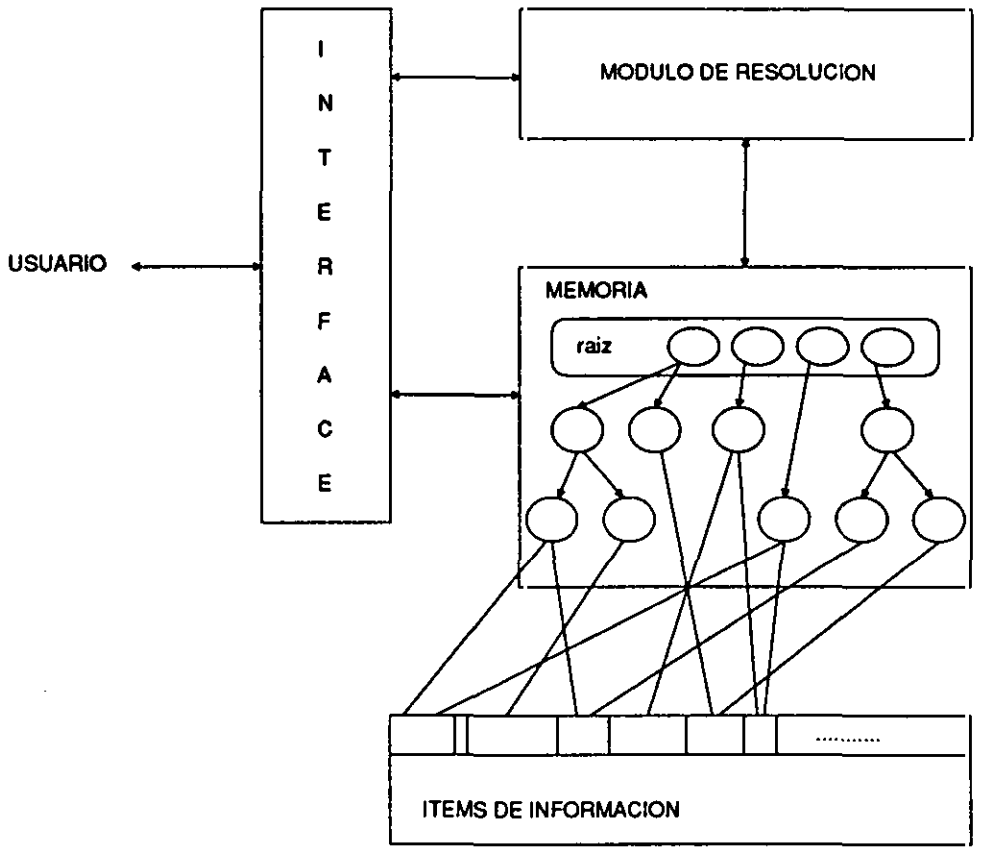

Figura 3.1

es lo que tomamos de Fernández-Valmayor (hay que notar que aunque la estructura de la memoria sea la misma, sus contenidos serán distintos). El bloque rotulado como "MODULO DE RESOLUCION" es el objetivo principal de nuestro trabajo.

Dedicamos este capítulo a describir la memoria y los procesos necesarios para mantenerla.

3.2 EL SISTEMA

3.2.1 Estructura de la información del sistema: CDs

En este modelo de memoria, las unidades básicas de conocimiento son las estructuras de la Dependencia Conceptual (CDs)(Schank, 1972). Estas constan de una "cabeza" con una lista de propiedades y valores. En la literatura de los sistemas basados en el conocimiento, estas propiedades -según la información que contengan y el contexto- han recibido otros nombres como atributos, "roles" o "slots". Los valores de estas propiedades se ha denominado con frecuencia "fillers".

En principio, se consideran tres tipos básicos de "cabezas": ACCION, ESTADO y PP, éste último significando los objetos físicos o "picture producer" que Schank propone en su teoría de la teoría de la Dependencia Conceptual. En nuestro sistema, aplicado a la síntesis de programas, con los CDs ACCION y PP representamos, respectivamente, las operaciones que se llevan a cabo en cualquier programa, y los objetos (datos estándares o tipos abstractos de datos) sobre los que se realizan dichas operaciones. Con ESTADO, representamos el estado de los objetos; sobre estos estados se pregunta en las condiciones de los programas.

El significado de un texto no puede expresarse, en general, utilizando sólo estos 3 elementos básicos. Normalmente un texto incluye además una serie de relaciones: temporales, causales, etc. entre los elementos (oraciones) que lo componen. Para representar esta información, a la que llama "información compleja", se utilizan dos formalismos diferentes.

El primero de estos formalismos consiste en representar la información compleja mediante una red, en cuyos nodos tenemos CDs y cuyos arcos (*links*) representan las diferentes relaciones, o enlaces, posibles entre ellos: causales, temporales, espaciales etc.

El segundo lo constituyen los CDs complejos, es decir, CDs en los que los valores (*fillers*) que toman sus propiedades son a su vez CDs.

La posibilidad de convertir una estructura del primer tipo en otra del segundo y viceversa, además de plantear diversas cuestiones teóricas - que Fenández-Valmayor (1990) expone-, presenta la posibilidad de ver un mismo CD complejo o red desde distintos puntos de vista, lo que permite, por ejemplo, ver una acción desde el punto de vista de la acción en si misma, o desde el punto de vista del sujeto, o desde el punto de vista del objeto que sufre la acción.

Para representar la información, se definen dos tipos de estructuras, una para los CDs y otra para los LINKs (enlaces o relaciones entre diferentes CDs).

La estructura con la que se implementan los CDs consta de tres campos: HEAD, FEATURES y R-LINKS.

El campo HEAD es un átomo que representa la clase de aserción que hace este CD (si se trata de una acción, un estado o un PP). El campo FEATURES es la lista de propiedades y valores (implementada como una a-list de pares atributo-valor), y el campo R-LINKS, es una lista de estructuras del tipo LINK-R. Son enlaces que permiten relacionar este CD con otros. Los campos que componen un LINK-R son:

- TYPE, es un átomo que identifica el tipo de la relación entre los CDs: temporal, estructural, etc.; y

- CD-R, que contiene el CD con el que se establece la relación.

Hay que tener en cuenta que el valor o *filler* de una propiedad es a su vez otro CD, de forma que la definición de CDs es recursiva hasta que tenemos un CD que sólo tiene "cabeza", es decir, tiene su lista de propiedades vacía.

El hecho de que el "valor" que toma un atributo, sea a su vez un CD, nos permite representar sucesivos niveles de detalle en la información.

A continuación veremos algunos ejemplos (tomados de la cinemática y presentados por los propios autores del sistema que estamos comentando). Distinguiremos así, CD simple, CD complejo y red de CDs.

Antes diremos que estas estructuras (CD -simple o no- y LINK-R) son creadas por las funciones de entrada al sistema ('list-cd' y 'crea- link-r'; ver apéndice) a partir de la información de entrada. En el próximo capítulo nos detendremos en ellas con más detalle.

El siguiente CD representa el episodio: "Galileo lanza desde su casa un proyectil". (Se ha tomado de R. Schank la primitiva "propel": aplicar una fuerza a un objeto. Para otros ejemplos tomaremos las primitivas "grasp" y ptrans": asir o empuñar una cosa y cambiar de posición, respectivamente).

```
ACCION tipo PROPEL
       actor GALILEO
       objeto PROYECTIL
       desde SU-CASA
```

En este ejemplo, ACCION es la cabeza del CD; PROPEL es el *filler* del *slot* 'tipo', que a su vez es la cabeza de un CD cuya lista de propiedades es vacía. También son cabezas de CDs con una lista de propiedades vacía GALILEO, PROYECTIL, etc. Se trata, por tanto, de un CD simple.

En el siguiente ejemplo, el valor del *slot* "actor" y "objeto", son a su vez CDs, cuyas listas de propiedades no están vacías. Tenemos por tanto un CD complejo.

```
ACCION tipo PROPEL
       actor PP tipo HUMANO
                nombre GALILEO
                sexo VARON
                edad ADULTO
                profesion MEDICO
       objeto PP tipo  MASA-INERTE
                 nombre PROYECTIL
                 forma ESFERICA
       desde SU-CASA
```

El episodio representado es: "Galileo, un adulto de la especie humana de sexo varón y de profesión médico, lanza un proyectil, una masa inerte de forma esférica, desde su casa".

Como ya se expuso, el hecho de que el *filler* que toma un *slot* sea a su vez un CD, permite representar sucesivos niveles de detalle en la información, por ejemplo, especificar quién es Galileo o cuáles son las características del proyectil.

Siguiendo el ejemplo, veamos cómo se puede transformar este CD en una red. Se convierten los *slots* "actor" y "objeto" en enlaces (*links*), añadiendo el sufijo PP si el enlace señala al PP, y el sufijo A si el enlace señala a la acción.

Este cambio en la forma de representación es el que permite ver un mismo hecho desde distintos puntos de vista, teniendo un significado más profundo que una mera cuestión de estilo o notación. Podríamos por ejemplo, cambiar el foco de atención desde la "acción propel" al autor de la acción "Galileo" o al objeto lanzado.

Por último veamos un ejemplo de cómo suministrar al sistema una red de CDs. Supongamos que se quiere representar el episodio: "Galileo se traslada desde su casa a la torre de Pisa y desde allí deja caer un proyectil, que se mueve hacia el suelo con una velocidad creciente debido al impulso de la fuerza de la gravedad".

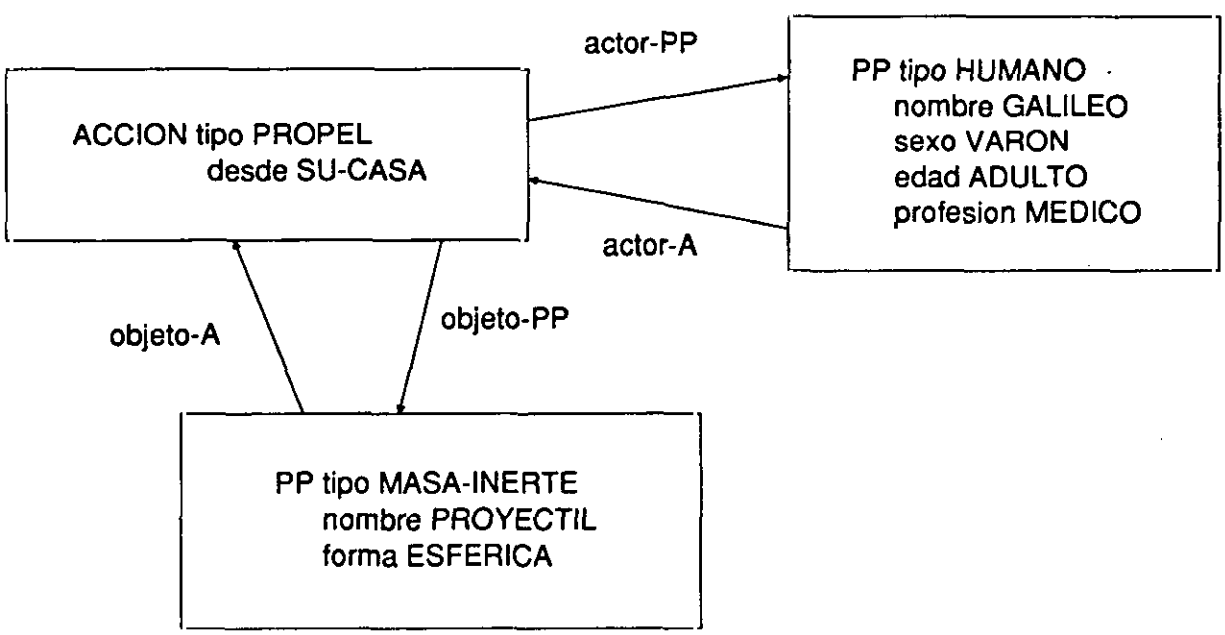

Figura 3.2

Para esto, se crean primero una serie de CDs, que representan los episodios de la historia completa, y a continuación, se relacionan los CDs entre si, por parejas, del modo que indica el texto.

Se establecen los siguientes CDs:

```
CD1:
      ACCION tipo PTRANS
             actor GALILEO
             objeto GALILEO
             desde SU-CASA
             hacia PP tipo TORRE
                      situacion PISA

CD2:
      ACCION tipo GRASP
             actor GALILEO
             objeto PROYECTIL
             desde TORRE
             valor NEGATIVO

CD3:
      ACCION tipo PTRANS
             actor PROYECTIL
             objeto PROYECTIL
             desde TORRE
             hacia SUELO
             velocidad CRECIENTE

CD4:
      ACCION tipo PROPEL
             actor GRAVEDAD
             objeto PROYECTIL
```

A continuación se establecen las relaciones temporales entre CDs:

```
CD1 va 'antes' que CD2
CD2 va 'antes' que CD3
CD2 va 'antes' que CD4
```

y las relaciones:

```
CD4 'es-causa' del CD3
CD2 'hace-posible' el CD4
```

Las relaciones creadas van en los dos sentidos, es decir, si se especifica que el suceso descrito por CD1 ocurre antes que el descrito por CD2, esta función creará -además del enlace "antes"- su inverso "después".

Como ya se ha dicho antes en este mismo apartado, la información compleja que representa una red, puede representarse mediante un CD complejo. Lo importante es que esta correspondencia (red - CD complejo), no es única. De hecho, existen tantas representaciones de una red en forma de CD complejo como nodos tenga la red. Dicho de otro modo, transformar una red en un CD complejo implica imponer un punto de vista sobre los acontecimientos que la red representa.

Veamos cómo se llevaría a cabo esta transformación en el ejemplo anterior. Primero, para convertir la red de CDs en un CD complejo, debemos tomar uno de sus nodos como punto de partida y fijar una dirección de recorrido de la misma. Después, creamos un nuevo tipo de "cabeza" distinto a los tres tipos básicos (acciones, estados y objetos o PPs), utilizando la "cabeza" del CD utilizado como punto de partida y finalmente añadimos a la lista de propiedades de cada CD los *links* que parten de él.

La figura 3.3 es la que se obtiene si tomamos como CD de partida a CD2 que expresa la idea de que Galileo deja caer el proyectil.

Lo que se pretende en este sistema con la utilización de las redes y los distintos puntos de vista es obtener más esquemas mentales, como veremos al hablar del tratamiento de estructuras de información complejas. Nosotros las utilizaremos para representar la solución de un programa, queriendo representar con esto las distintas acciones que hay que ejecutar para resolverlo así como la secuencialidad en que deben ejecutarse.

La posibilidad de establecer otros puntos de vista la utilizaremos sólo para ver la acción desde el punto de vista de los datos. De esta manera, como se verá más tarde, obtenemos más enlaces desde la definición de un tipo de datos en general, a los casos concretos en los que se utiliza. Sin embargo, este cambio de punto de vista, como se explicará en su momento, difiere -en alguno de sus aspectos- de cómo lo hacen Fernández-Valmayor y Fernández Chamizo.

3.2.2 Estructura de la memoria

Los CDs que acabamos de ver, no constituyen en si mismos la base de conocimientos del sistema. La "memoria" está constituida por una red jerarquizada en cuyos nodos tenemos unas estructuras denominadas MOPs (Memory Organizations Packets, Schank, 1982). Estas son creadas dinámicamente por el sistema a partir de la información recibida previamente (CDs). Su misión principal es la de indexar la información de entrada, al mismo

```
MOP-ACCION tipo GRASP
           actor GALILEO
           objeto PROYECTIL
           desde TORRE
           valor NEGATIVO

           antes ACCION tipo PTRANS
                        actor GALILEO
                        objeto GALILEO
                        desde CALLE
                        hacia PP tipo TORRE
                                 posicion PISA

           despues ACCION tipo PTRANS
                          actor PROYECTIL
                          objeto PROYECTIL
                          desde TORRE
                          hacia SUELO
                          velocidad CRECIENTE

           hace-posible ACCION tipo PROPEL
                               actor GRAVEDAD
                               objeto PROYECTIL

                               causa ACCION tipo PTRANS
                                            actor PROYECTIL
                                            objeto PROYECTIL
                                            desde TORRE
                                            hacia SUELO
                                            velocidad CRECIENTE
```

Figura 3.3

tiempo que forman contextos o generalizaciones que han de servir para interpretar dicha información.

La red jerarquizada que forma la memoria se constituye a partir de dos estructuras básicas: los MOPs y los LINK-S (Specialization Links), formando los primeros los nodos de la red y los segundos los arcos que conectan estos nodos entre sí, o con los nodos terminales que contienen los items de información. Estos items de información son las estructuras terminales a las que finalmente apunta la red jerarquizada que compone la

memoria; son las estructuras CD y LINK-R creadas por las funciones de entrada al sistema a partir de la información de entrada.

La composición de estas estructuras es la siguiente:

Un MOP tiene tres campos. El primero -PATTERN- apunta a su padre (el nodo padre en la jerarquía); CONTENTS, el segundo, es una estructura del tipo CD y constituye el contenido del MOP, y el último, S-LINKS, es una lista de estructuras LINK-S que sirven para apuntar a los MOPs o particularizaciones que especializan el contenido del MOP.

Una estructura LINK-S se define con los campos siguientes:

El campo TYPE, contiene un átomo que indica si la estructura a la que apunta es un sub-MOP (un MOP que especializa el MOP anterior) o una particularización (*instance*). Una particularización es un ítem de información, tal y como ésta es suministrada por los usuarios, constituido por un CD o una red de CDs).

CHILD contiene el sub-MOP o ítem de información (*instance*) apuntado.

SLOT contiene el atributo por el cual indexamos. Corresponde a una diferencia entre el contenido del sub-MOP y el contenido del MOP padre.

CD contiene el valor o *filler* del atributo señalado en SLOT.

Finalmente, un contador, COUNT, inicializado a cero que se utilizará para reforzar o debilitar el LINK-S.

Esta forma de organización de la memoria sigue, en líneas generales, el modelo utilizado por los "Sistemas basados en la memoria" tal como se ha descrito en capítulos anteriores. Recordemos brevemente que, en este esquema de organización, una particularización (un ítem de información), o un sub-MOP (una estructura de información creada por el sistema generalizando varias particularizaciones), se indexa bajo un MOP padre mediante todas aquellas características que lo diferencian de él. De esta forma se crea una rica estructura de índices que permite acceder a la información buscada, sin necesidad de explorar secuencialmente todos los items contenidos en la memoria, y facilita al mismo tiempo la creación de nuevos sub-MOPs, mediante el proceso de generalización que más adelante describiremos con detalle.

La memoria del sistema es, por tanto, una estructura jerárquica. En su raiz tenemos inicialmente una lista con los MOPs que se consideran como formalismo básico de representación y cuyos contenidos son los CDs:

```
cabeza:       ACCION      cabeza:       PP          cabeza:       ESTADO
propiedades:  NIL         propiedades:  NIL         propiedades:  NIL
r-links:      NIL         r-links:      NIL         r-links:      NIL
```

Estos MOPs constituyen la semántica inicial del sistema. A partir de aquí, el prototipo va indexando toda la información que recibe, formando para ello nuevos sub-MOPs que permiten referenciar ésta por cada una de sus características.

3.2.3 Procesos de incorporación de la información a la memoria

A continuación vamos a ver los procesos que incorporan información al sistema y las consecuencias que estas operaciones tienen sobre la memoria.

Recordemos que se consideran dos tipos de información: unidades de información simples, formadas por una única estructura de dependencia conceptual o CD, y las unidades de información complejas, formadas por redes de CDs relacionados entre sí.

Primero veremos el proceso de incorporación de unidades de información simples. Las operaciones realizadas con estas estructuras, servirán como base para describir la incorporación a la memoria de estructuras complejas. Estos procesos los describiremos muy brevemente ya que nuestro sistema no hará uso de ellos, ya que el objetivo que persigue es distinto.

3.2.4 Tratamiento de las estructuras de información simples

En este apartado vamos a ver cómo procesa el sistema unidades de información simples, es decir, CDs que son tratados como un único bloque de información aunque tengan más de un nivel de profundidad.

El sistema ante una entrada de estas características, busca primero en la memoria estructuras similares a la dada; después construye índices que le permiten referenciar el nuevo ítem de información desde las estructuras similares halladas y finalmente crea, si es posible, nuevas generalizaciones y reorganiza la memoria para darles acomodo en ella.

Compatibilidad entre CDs. Búsqueda de información similar.

El método de búsqueda en la memoria de situaciones similares a la información recibida se basa en el concepto de compatibilidad entre CDs. Dos CDs son compatibles si uno puede ser considerado como especialización del otro, es decir, si tienen la misma cabeza y sus propiedades comunes tienen idéntico valor. Siguiendo los mismos ejemplos dados por los autores, los CDs

```
ACCION tipo PROPEL
       actor GALILEO
       objecto PROYECTIL
```

```
ACCION tipo PROPEL
       actor GALILEO
       objeto PROYECTIL
       desde TORRE
       hacia SUELO
```

son compatibles ya que las propiedades o atributos comunes, tipo, actor y objeto, tienen el mismo valor (*filler*). De hecho, en la segunda estructura lo único que hacemos es añadir detalles, que no contradicen la información representada en la primera.

Por el contrario, estos otros CDs no serían compatibles, ya que difieren por su cabeza y/o en el valor de los atributos de igual nombre.

```
ACCION tipo PROPEL          ACCION tipo PROPEL          PP tipo PROYECTIL
       actor GALILEO                actor ANDREA            masa PESADO
       objeto PROYECTIL             objeto PROYECTIL        forma ESFERICO
                                    desde TORRE
                                    hacia SUELO
```

Un caso más interesante se plantea al analizar la compatibilidad de los CDs anteriores con un CD como el siguiente:

```
ACCION tipo PROPEL
       actor PP tipo HUMANO
                nombre GALILEO
                profesion MEDICO
       objeto PROYECTIL
```

En este ejemplo utilizamos la recursividad de los CDs para especificar, mediante otro CD, *qué debe entender el sistema por GALILEO: un ser humano de profesión médico.*

La dificultad estriba en que este último CD no sería compatible con los anteriores al no coincidir la cabeza del CD que está en el *slot* "actor" del primero (GALILEO), con la cabeza del *slot* "actor" del último CD (PP) lo cual no tendría sentido, ya que en este último caso lo único que hacemos es añadir nueva información sobre Galileo sin contradecir la anterior.

Es lógico que si un CD es una estructura de datos recursiva, la función que comprueba la recursividad entre CDs también lo sea, es decir, que se llame a sí misma para comprobar que los valores de dos atributos o propiedades con el mismo nombre son compatibles entre sí. Aún así, este esquema fallaría en casos como el anterior. Para resolver esta dificultad, se introduce la idea de que la cabeza de un CD puede ser no solamente una de las clases ACCION, ESTADO y PP, sino también un símbolo que, como GALILEO, representa una realidad (una acción, un estado o un PP) que puede ser especificada posteriormente.

En consecuencia, para comprobar la compatibilidad entre el símbolo GALILEO, representado como un CD cuyo único elemento distinto de NIL es su cabeza, y otro CD, tendremos que comprobar si este símbolo coincide con el valor de alguno de los *slots* del

segundo CD que, como "nombre", pueden ser utilizados para representar toda la estructura.

A partir de esta noción de compatibilidad, se desarrolla el sistema de búsqueda de estructuras similares a la información de entrada. Esta comienza a partir de la raíz de la memoria que -como ya hemos visto- es -inicialmente- una lista con los tres MOPs básicos.

Para llevar a cabo esta tarea, se define una función de búsqueda que trata de encontrar en la memoria todos los MOPs compatibles con su argumento (el CD que representa la información que tratamos de incorporar) y que sean terminales.

Un MOP es compatible con un CD, si dicho CD y el CD que ocupa el campo CONTENTS del MOP son compatibles. Un MOP es terminal si no tiene indexados nuevos sub-MOPs o, si los tiene, no son compatibles con el CD inicial.

La función de búsqueda comienza seleccionando los MOPs compatibles que haya en la raiz de la memoria, y va siguiendo los LINK-S de los sub- MOPs compatibles hasta encontrar un MOP terminal. Finalmente la función devuelve una lista con todos los MOPs encontrados y que sirven para interpretar (desde la óptica del sistema) la nueva información.

La búsqueda en memoria tomando como patrón un CD puede tener diversos fines: indexar ese CD en memoria o simplemente buscar la información similar a ese CD que pueda haber en memoria.

3.2.5 Creación de índices y nuevos sub-MOPs.

Incorporar un nuevo ítem de información a la red que conforma la estructura de la memoria requiere varios pasos. En el epígrafe anterior hemos visto cómo se realiza el primero de ellos. Al final, la función de búsqueda entrega una lista con los MOPs en memoria que contienen generalizaciones compatibles con el nuevo ítem de información. A partir de aquí el sistema procede de la siguiente manera:

En un primer paso, crea los índices (LINK-S) que le permitirán referenciar la nueva información desde los nodos de la red de la memoria localizados en el proceso anterior de búsqueda (MOPs compatibles).

En segundo lugar, crea (si es posible) nuevos nodos (sub-MOPs) que generalizan el contenido de la nueva particularización introducida y de otras similares indexadas en operaciones anteriores. A continuación, incorpora estos sub-MOPs a la estructura de la memoria, indexándolos en la red, y reindexando los CDs utilizados para obtener la generalización bajo los nuevos sub-MOPs.

En el primer paso, y por cada uno de los MOPs compatibles, se crean nuevas estructuras del tipo LINK-S correspondientes a cada una de las propiedades (atributos o *roles*) de la nueva información que no están en el MOP padre. Con toda esta información se completan los *slots* de LINK-S y por último, esta estructura se coloca en el campo S-LINKS del MOP padre.

Para ilustrar mejor cuál es el efecto que produce en la memoria el proceso de generalización vamos a ver cómo funciona mediante un ejemplo.

La figura 3.4, muestra cuál es la situación de la memoria en el nodo que representa la generalización de las "acciones que implican un cambio de posición".

Esta generalización está contenida en un MOP (el MOP padre), que tiene indexados por debajo de él tres episodios. El índice de cada episodio está constituido por una estructura LINK-S que señala, mediante SLOT y CD, una de las diferencias con el MOP padre: la correspondiente a "objeto". (Además de los índices que se muestran en la figura, habría otros que también indexarían estos episodios mediante el índice correspondiente al *slot* "actor").

Puede observarse que cada índice está formado tanto por el *slot* como por el *filler* de la diferencia entre el episodio y la generalización contenida en el MOP padre.

Tenemos por tanto, tres episodios indexados por el mismo *slot* "objeto", donde dos de ellos tienen un *filler* compatible. El CD

```
LIBRO
```

es compatible con

```
PP nombre LIBRO
   autor COPERNICO
```

Para éstos, puede iniciarse el proceso de generalización.

El resultado de dicho proceso se muestra en la figura 3.5: bajo el MOP que contiene la generalización "acciones que implican un cambio de posición", se ha formado una nueva generalización que es una especialización de la anterior: "acciones que implican un cambio de posición de un PP de nombre 'libro' y autor 'copérnico'".

Esta especilización está indexada por su diferencia, el *slot* "objeto". Bajo ella se han re-indexado los dos episodios que sirvieron para crearla, utilizando otra vez como índice la diferencia con su MOP padre, en este caso marcada por el *slot* "actor".

La generalización formada en el ejemplo anterior es quizá muy específica. Implica cambiar de posición un PP que sea un libro cuyo autor es Copérnico. Por tanto, podría

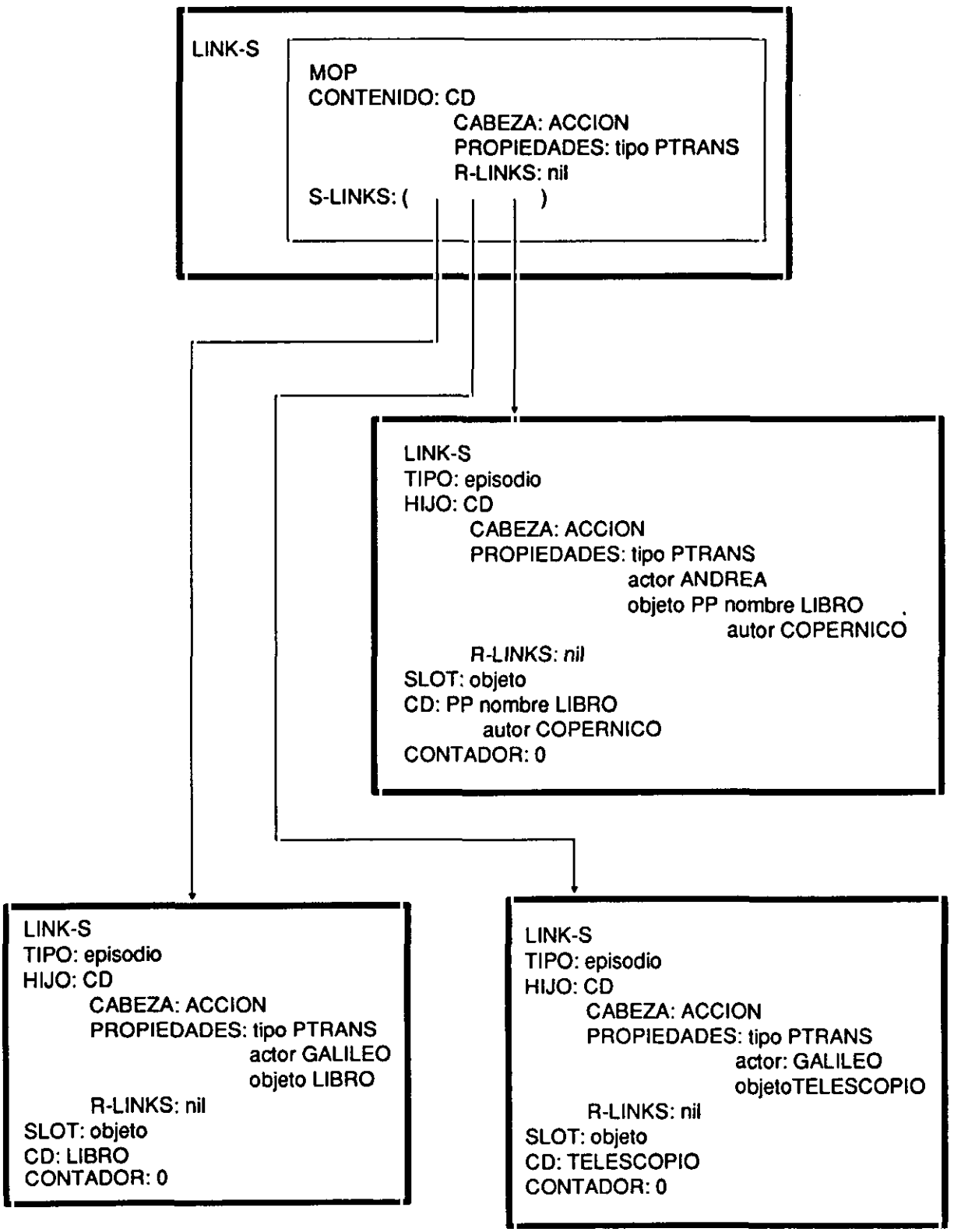
LINK-S
MOP
CONTENIDO: CD
CABEZA: ACCION
PROPIEDADES: tipo PTRANS
R-LINKS: nil
S-LINKS: ( )
LINK-S
TIPO: episodio
HIJO: CD
CABEZA: ACCION
PROPIEDADES: tipo PTRANS
actor ANDREA
objeto PP nombre LIBRO
autor COPERNICO
R-LINKS: nil
SLOT: objeto
CD: PP nombre LIBRO
autor COPERNICO
CONTADOR: 0
LINK-S
TIPO: episodio
HIJO: CD
CABEZA: ACCION
PROPIEDADES: tipo PTRANS
actor GALILEO
objeto LIBRO
R-LINKS: nil
SLOT: objeto
CD: LIBRO
CONTADOR: 0
LINK-S
TIPO: episodio
HIJO: CD
CABEZA: ACCION
PROPIEDADES: tipo PTRANS
actor: GALILEO
objetoTELESCOPIO
R-LINKS: nil
SLOT: objeto
CD: TELESCOPIO
CONTADOR: 0

Figura 3.4

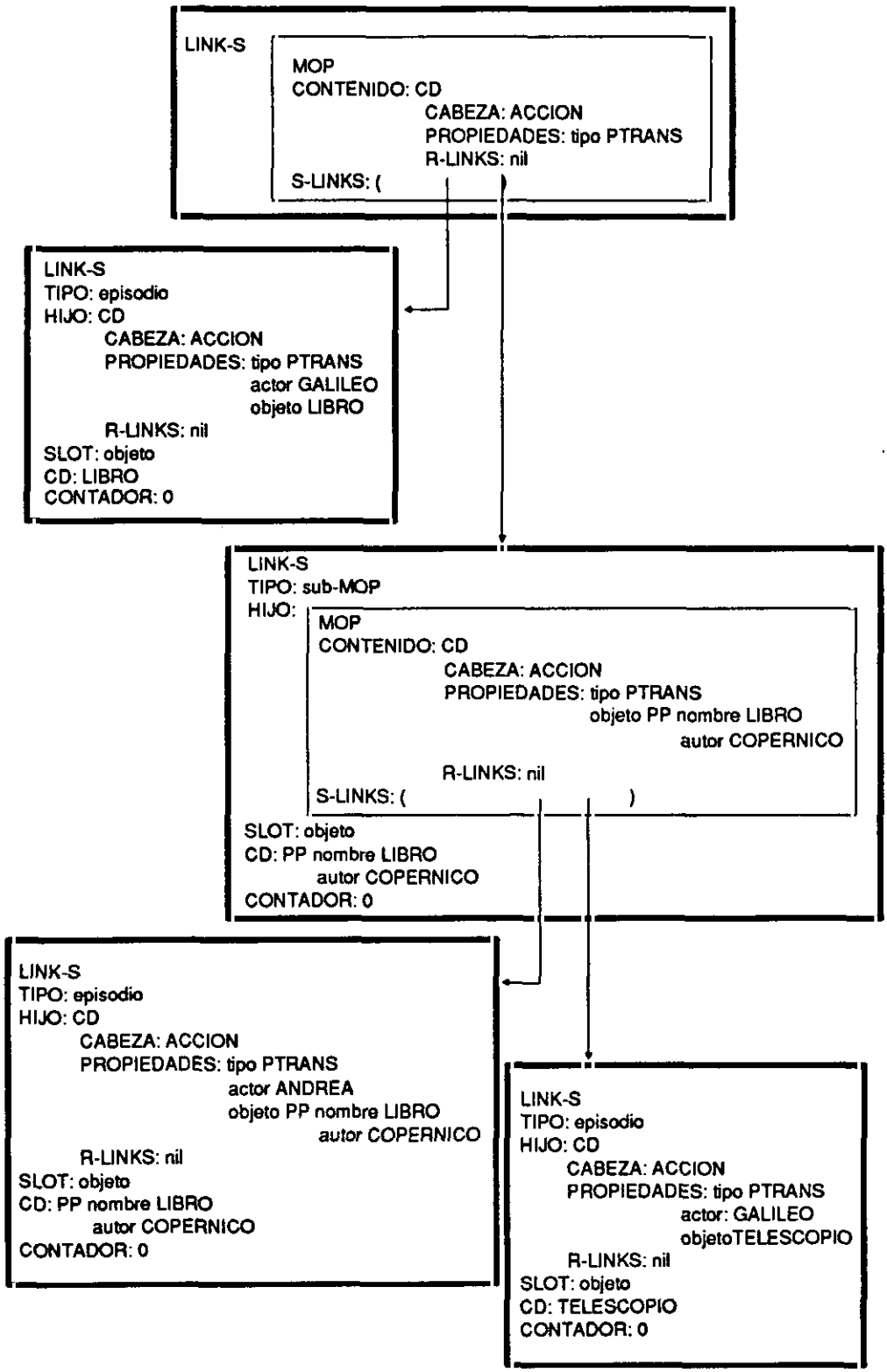
LINK-S
MOP
CONTENIDO: CD
CABEZA: ACCION
PROPIEDADES: tipo PTRANS
R-LINKS: nil
S-LINKS: (
)
LINK-S
TIPO: episodio
HIJO: CD
CABEZA: ACCION
PROPIEDADES: tipo PTRANS
actor GALILEO
objeto LIBRO
R-LINKS: nil
SLOT: objeto
CD: LIBRO
CONTADOR: 0
LINK-S
TIPO: sub-MOP
HIJO:
MOP
CONTENIDO: CD
CABEZA: ACCION
PROPIEDADES: tipo PTRANS
objeto PP nombre LIBRO
autor COPERNICO
R-LINKS: nil
S-LINKS: (
)
SLOT: objeto
CD: PP nombre LIBRO
autor COPERNICO
CONTADOR: 0
LINK-S
TIPO: episodio
HIJO: CD
CABEZA: ACCION
PROPIEDADES: tipo PTRANS
actor ANDREA
objeto PP nombre LIBRO
autor COPERNICO
R-LINKS: nil
SLOT: objeto
CD: PP nombre LIBRO
autor COPERNICO
CONTADOR: 0
LINK-S
TIPO: episodio
HIJO: CD
CABEZA: ACCION
PROPIEDADES: tipo PTRANS
actor: GALILEO
objetoTELESCOPIO
R-LINKS: nil
SLOT: objeto
CD: TELESCOPIO
CONTADOR: 0

Figura 3.5

suveder que en el futuro, el sistema no recibiera más información compatible con esta generalización y, por consiguiente, que no se indexaran nuevos episodios por debajo de ella.

Cada vez que el sistema accede a un MOP, comprueba si la nueva información recibida es compatible con la generalización contenida en él. Si el resultado es negativo, el campo COUNT de la estructura LINK- S del índice correspondiente se debilita restándole uno. Pasado cierto umbral, los episodios dejan de ser accesibles mediante ese índice. Esta operación no implica que la información se borre de la memoria, sino simplemente, que perdemos uno de los múltiples índices que nos permiten acceder a esa información. Eventualmente, la información quedaría definitivamente perdida cuando no estuviese activo ninguno de los índices que nos permiten acceder a ella.

De esta forma, el sistema mantiene bajo control el crecimiento de la memoria, al mismo tiempo que se simula el fenómeno del olvido en la memoria humana.

3.2.6 Tratamiento de las estructuras de información complejas

Desde los trabajos pioneros de Barlett (1932) se admite que la memoria humana se vale de algún tipo de "esquema", adquirido en experiencias anteriores, para interpretar la información que recibe.

Puesto que se pretende obtener un modelo cognitivo de la memoria humana, es necesario sacar el máximo número de "esquemas" posibles de un red de CDs. Esto se consigue pasando de una red de CDs a un CD complejo imponiendo puntos de vista. Se pueden obtener tantos CDs complejos como nodos tiene la red, y por cada CD complejo obtener un "esquema".

Procesar la información contenida en una red de información compleja requiere la ejecución de procesos en cada uno de los nodos de la misma.

El primer proceso consiste en indexar individualmente el CD contenido en cada uno de los nodos, siguiendo el procedimiento descrito anteriormente para los items de información simples.

El segundo proceso implica la creación de estructuras complejas que representan toda la información contenida en la red desde diferentes perspectivas o puntos de vista, que son la base para obtener nuevas estructuras de conocimiento que representen la información a un nivel de abstracción superior.

Lo que se obtiene mediante el primer proceso es la posibilidad de poder referenciar la red de CDs desde múltiples puntos de la memoria, representando cada uno de estos puntos la generalización obtenida con cada uno de los CDs individuales.

El segundo proceso implica convertir una red en una estructura jerárquica. Para ello fija un recorrido de la misma, partiendo del nodo que se ha indexado en el primer paso, y que se considera como CD principal. A continuación se integran todos los CDs que componen la red en una única estructura, un CD complejo, convirtiendo los LINK-R (relaciones) entre CDs en *slots* del CD complejo.

El resultado de este proceso es que las generalizaciones que se pueden obtener de este CD complejo son diferentes de las que se pueden obtener de los otros CDs complejos (obtenidos de la misma red sin más que tomar como punto de partida otro nodo de la misma). Es decir, la misma red de información, proporciona esquemas (causales, temporales, etc.) distintos dependiendo del punto de vista. No realizar este proceso con la red implicaría perder la posibilidad de encontrar esquemas que pueden tener una gran capacidad para organizar la información en un determinado dominio de conocimiento.

En nuestro dominio concreto, no necesitamos obtener estos esquemas desde tantos puntos de vista. Sería tanto como querer tener esquemas de algoritmos vistos no como un todo que resuelve un problema, sino como sentencias a ejecutar unas delante de otras sin dejar claro qué se pretende resolver. De los distintos puntos de vista posibles desde los cuales se puede ver un CD, en nuestro sistema se describirán desde el punto de vista de la acción.

Abstracción de conceptos por sustituición de variables

En el modelo de memoria que estamos describiendo, la abstracción de conceptos generales a partir de CDs complejos tiene dos etapas. La primera consiste en utilizar básicamente los mismos procesos de generalización utilizados para los CDs simples. Es decir, formar nuevos sub-MOPs extrayendo las características comunes de los episodios indexados -mediante índices compatibles- bajo el mismo MOP padre.

La segunda consiste en sustituir por variables aquellos CDs que aparecen repetidamente como el valor (*filler*) del mismo *slot* dentro del CD complejo. De esta forma se consigue ganar en generalidad y se establece de forma más específica la relación entre los diversos componentes del CD complejo.

Estas variables se han denominado pc-var (*predicate calculus* variables). Distinguiremos estas variables poniendo como caracter inicial una interrogación cerrada ("?"). Estas variables, que son realmente CDs, tienen vacia la lista de pares atributo-valor.

En algunos procesos (por ejemplo al comprobar la compatibilidad entre dos CDs) y para la primera aparición de la variable, necesitaremos ligarla a valores concretos. La restricción que se pone es que las sucesivas apariciones de esa variable deben unificar con el primer valor emparejado.

El ejemplo que mostramos a continuación representa que "la acción de una fuerza sobre un proyectil, hace que éste se ponga en movimiento".

```
ACCION tipo PROPEL
       objeto PROYECTIL
       causa ESTADO tipo CINEMATICO
                    objeto PROTECTIL
                    velocidad NULA
                    después ESTADO tipo CINEMATICO
                                   objeto PROYECTIL
                                   velocidad CRECIENTE
```

por el segundo proceso de abstracción, vemos que PROYECTIL aparece repetidamente como "objeto". Generalizando, podemos sustituir PROYECTIL por una variable pc-var. El CD que se obtiene después de la generalización sería el siguiente:

```
ACCION tipo PROPEL
       objeto ?OBJETO
       causa ESTADO tipo CINEMATICO
                    objeto ?OBJETO
                    velocidad NULA
                    después ESTADO tipo CINEMATICO
                                   objeto ?OBJETO
                                   velocidad CRECIENTE
```

Este CD representa que "la acción de una fuerza sobre un objeto hace que dicho objeto se ponga en movimiento".

En nuestro sistema, utilizaremos estas variables para obtener patrones de programas. Como puede intuirse, estos patrones no se dan a priori, sino que es el propio sistema quien los genera.

A diferencia de lo que aquí se ha expuesto, estas variables, no las obtendremos por exploración dentro de un único CD complejo, sino que estas variables se establecerán entre los nodos que compongan una parte determinada de una red de CDs. En el próximo capítulo explicaremos qué parte de la red se toma y por qué sólo esa parte.

La memoria que hemos desrito constituye la base de conocimientos del sistema CBR que presentaremos en el siguiente capítulo. Esta memoria es dinámica, permitiendo la posibilidad de añadir, en cualquier momento nuevos conocimientos así como la capacidad de aprendizaje del sistema.

4. NUESTRO SISTEMA

En el capítulo anterior hemos descrito el sistema de memoria del que partimos en nuestro trabajo y que contendrá todo el conocimiento -es la base de conocimientos- necesario para los métodos de inferencia que proponemos.

Muchos trabajos de programación automática se han enfocado hacia el papel que juega la deducción y el conocimiento general de programación en el desarrollo de programas. Sin embargo, en este campo de la inteligencia artificial -como en muchos otros- el papel que juega el conocimiento del dominio es crucial. El sistema necesita este conocimiento ya que durante la interacción con el usuario no sólo comparte nociones -conceptos- y notaciones, sino también una gran cantidad de detalles del conocimiento concreto del dominio sobre el que trabaja (Barstow, 1984).

Todos estos detalles de conocimiento y nociones necesarias, se los proporcionamos al sistema por medio de un diccionario de conceptos. Estos conceptos, se enlazarán -como veremos más adelante, y a medida que se vayan suministrando al sistema- con los problemas resueltos, que son casos concretos.

Una vez la memoria tiene los conocimientos necesarios y una biblioteca de casos suficientemente amplia, el usuario puede plantear nuevos problemas al sistema.

El usuario especifica el problema a resolver a través de un "parser", idealmente en lenguaje natural. Esta información se presenta al razonador basado en casos para que obtenga una posible solución.

El sistema trata los nuevos casos buscando analogías con los que tiene almacenados en su biblioteca. Partiendo del caso o casos más análogos y ayudado del diccionario de conceptos, el sistema presentará una posible solución.

Antes de describir, con mayor profundidad, los procesos citados anteriormente, vamos a presentar una panorámica general de la arquitectura del sistema.

4.1 ARQUITECTURA GENERAL DEL SISTEMA

La figura 4.1 ilustra el marco general que proponemos como un entorno para la síntesis de programas basada en casos (Méndiz *et al.*, 1992).

La *memoria* consta de dos componentes: un conjunto de conceptos - que junto con una tabla externa se convierte en un diccionario de conceptos- y la biblioteca de casos resueltos -los que se dan como casos de entrenamiento al sistema y los que resuelve él mismo-.

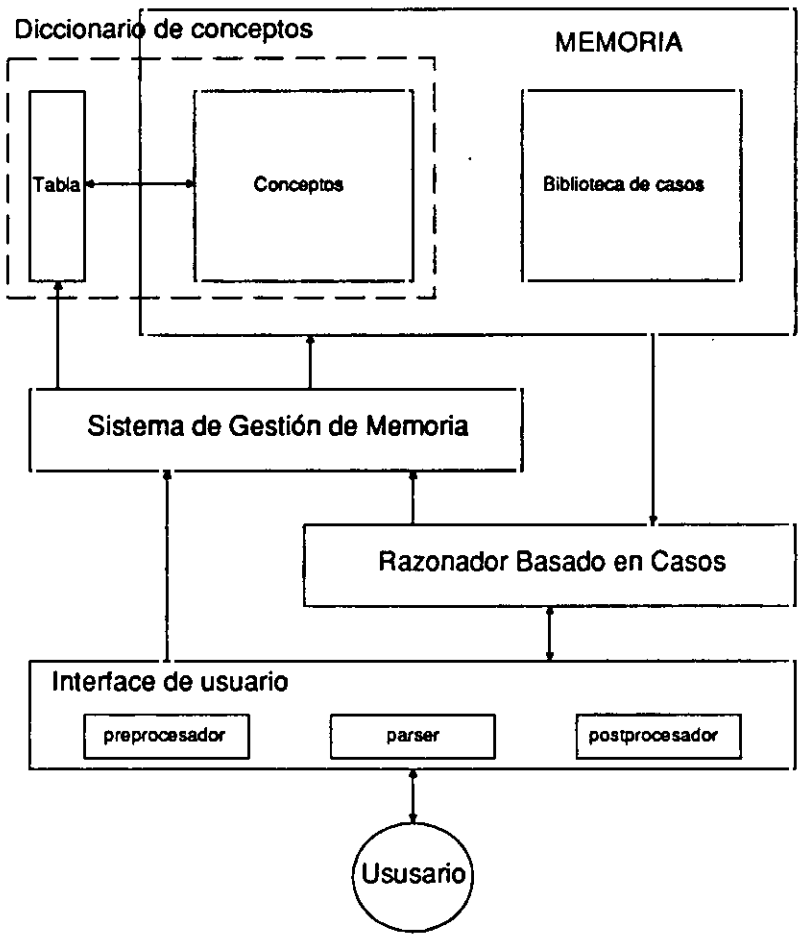

Figura 4.1

El diccionario de conceptos contiene el vocabulario necesario para especificar los problemas de un dominio determinado de forma natural y sucinta. Contiene información sobre los objetos y funciones dependientes del dominio que son útiles para la especificación de los problemas.

La biblioteca de casos contiene los problemas resueltos que se le han dado al sistema como casos de entrenamiento, así como los resueltos por él mismo. Además, con el tiempo, el sistema genera "patrones" y "esquemas" que también se guardan en esta biblioteca. Los "patrones" son métodos abstractos de resolución; los "esquemas" consisten en las operaciones comunes a un conjunto de programas relacionados entre sí.

Como su nombre indica, el *sistema de gestión de memoria* es el encargado de mantener clasificada y jerarquizada toda la información: los conceptos y los casos resueltos. Recibe información del usuario, cuando éste aumenta el diccionario de conceptos o la biblioteca

de casos, y del razonador cuando éste obtiene la solución a un nuevo problema, ya que además de presentarla al usuario que la solicitó, la guarda en la memoria.

El *razonador basado en casos* utiliza la información de la memoria para resolver un nuevo problema según las especificaciones que le presenta el usuario a través de la interfaz.

Esta *interfaz del usuario* tiene varias funciones. El "parser" transforma la información dada por el usuario a estructuras CD (de las que hemos hablado anteriormente). Si se trata de nuevos casos de entrenamiento al sistema, pasan por el preprocesador y de éste al sistema de gestión que es quien lo almacena en la memoria. Cuando la solución va en dirección contraria, es decir, el sistema presenta una solución solicitada por el usuario, ésta pasa por el posprocesador.

Nuestro trabajo se ha centrado en el desarrollo del razonador basado en casos y del sistema de gestión de memoria. La interfaz de usuario es rudimentaria, y el usuario debe comunicarse con el sistema con estructuras CD.

El dominio concreto que hemos tomado como aplicación de nuestro sistema es el comprendido en un curso básico de estructuras de datos, siguiendo de alguna forma el proceso que se sigue para enseñar a programar a los alumnos. Así pues nos vamos a centrar en resolver ejercicios de programación que son habituales en cursos de estas características.

Generalmente, para diseñar un sistema experto se necesita el trabajo de un analista del dominio -familiarizado con los problemas de un dominio concreto-. El analista conoce los tipos de objetos, términos, operaciones y funciones que normalmente se usan para especificar problemas en un nivel abstracto de ese dominio en particular. También conoce las técnicas de resolución estándares de esos problemas, y herramientas disponibles para resolverlos. El ingeniero del conocimiento complementa el papel del analista; su conocimiento versa sobre técnicas de programación independientes del dominio, algoritmos, metodologías de diseño, tipos de datos, etc. El conocimiento que proporcionan el analista y el ingeniero se codifica en la base de conocimientos del sistema.

Una de las razones por la que hemos escogido el dominio descrito, es porque nos es familiar, y por tanto, podemos simular los papeles del analista del dominio y del ingeniero de conocimiento. Además, consideramos que es un dominio adecuado, puesto que, si se pretende explicitar en el sistema los conocimientos utilizados por los programadores, es necesario comenzar con las técnicas básicas de programación.

4.2 ESTRUCTURA DE LA INFORMACION SUMINISTRADA AL SISTEMA

En este apartado describiremos los distintos estados por los que pasa la memoria hasta estar en una situación en la que pueda hacer frente a la resolución de un problema.

Inicialmente, la memoria contiene los dos MOPs básicos de los que ya se ha hablado: ACCION y OBJETO. (A partir de ahora, para facilitar las cosas, hablaremos de OBJETO en vez de PP).

Para inicializar la memoria con estas dos estructuras preliminares se ha definido la función 'init-memory', que realiza los siguientes procesos: Primero, crea dos estructuras tipo CD con cabezas: ACCION y OBJETO, respectivamente, y con una lista de propiedades nula. A continuación crea dos estructuras de tipo MOP en cuyo campo CONTENTS se situa cada uno de los CDs anteriores. Por último, forma una lista con esos dos MOPs y la liga a la variable dinámica *TOP*, que de este modo, se constituye en la raíz de la memoria.

Estos MOPs constituyen la semántica inicial del sistema. Una vez establecida esta semántica inicial, nuestro prototipo va indexando toda la información que recibe, formando para ello nuevos sub-MOPs que nos permitan referenciar ésta por cada una de sus características.

El diccionario de conceptos y, a continuación, los casos de entrenamiento constituyen la información que se suministra al sistema. Este orden no impide que, en cualquier momento, se puedan suministrar más conceptos al diccionario. En principio, es lógico que se introduzca antes un concepto que un caso concreto que lo utilice. Si en algún momento se aporta el caso antes del concepto, el sistema lo detecta y pide al usuario que defina el concepto.

4.2.1 Dicionario de conceptos

La base de un sistema de programación automática que pueda comunicarse con el usuario en el nivel del dominio de la aplicación, está en una comprensión profunda de distintos conceptos -dependientes e independientes del dominio- y de sus interrelaciones. Este conocimiento debe estar disponible de forma explícita en el sistema. Estos conocimientos explícitos, suministran la semántica básica para razonar con una especificación de entrada y obtener la solución final ejecutable.

En nuestro sistema tenemos conceptos de dos tipos: los que se introducen directamente, y los conceptos que forma el propio sistema como abstracción de características comunes de un conjunto de conceptos.

Los conceptos introducidos directamente por el usuario, pueden ser nociones abstractas o bien conceptos concretos, por lo que se representarán por MOPs o CDs respectivamente. Estos MOPs y CDs se indexan, como se detallará más adelante, dentro de la organización jerárquica de los conceptos en la memoria.

Los conceptos formados por el sistema, por tratarse de abstracciones, se representarán siempre por MOPs. Es el propio sistema quien los crea y los indexa en su memoria.

Como ya se apuntó en el capítulo precedente al hablar de la organización de la memoria, y en concreto de las estructuras de entrada, aquí tenemos una diferencia con el sistema de memoria de Fernández-Valmayor (1991). En éste se suponía que el sistema sólo iba a recibir episodios concretos, *instances*, CDs; en nuestro caso, admitimos también como entrada ideas o conceptos que son ya en si mismos abstracciones -MOPs-, como son las nociones abstractas del diccionario de conceptos.

Para el dominio de trabajo elegido -problemas elementales de estructuras de datos- las nociones que incluimos en nuestro diccionario son las referentes a acciones u operaciones (como puede ser contar o recorrer), y objetos sobre los que se pueden efectuar dichas operaciones (por ejemplo lista o árbol). En su representación en CDs, se trataría de ACCIONES y OBJETOS respectivamente.

Un CD se compone de una cabeza, y de una lista de pares atributo- valor. En los CDs OBJETO (esta sería su cabeza) la lista de pares describe las características de cada objeto, muy dispares unas de otras y difíciles de delimitar a priori.

La lista de pares atributo-valor de las ACCIONES consiste en una lista de los distintos casos gramaticales que complementan al verbo. A cada caso gramatical le corresponde un mismo *slot*. Para que la lectura de un CD sea intuitiva, hemos escogido los siguientes: Complemento directo "objeto", complemento indirecto "a", complemento de nombre "de", complemento circunstancial (cc.) de modo "modo", cc. de cantidad "cantidad", cc. de origen "desde", cc. de lugar "en", etc.

Una acción diferenciada es la que consiste en determinar el estado de un cierto objeto; lo que comúnmente en programación se conoce como condiciones. En el nivel más alto de abstracción, a esta acción la hemos denominado PREGUNTAR. Su "objeto" será siempre un ESTADO que se afirma o niega: "'algo' es/no es 'algo'". Constará de dos *slots*: "sujeto" (aquello de lo que se afirma o niega) y "es" o "noes" (cuyo *filler* corresponde al predicado nominal que se afirma o niega respectivamente).

Si en alguna ocasión se presenta una ACCION como *filler* del *slot* "objeto", significa que debe entenderse como objeto el resultado de esa acción. Un ejemplo típico serían los resultados de operaciones como sumar, restar, etc.

Tal como describimos en el capítulo anterior, en la memoria del sistema, la localización de un CD o MOP -y por tanto de un concepto- se hace a través de sus atributos, buscando un CD compatible. En nuestro sistema, necesitamos que los conceptos no sólo estén accesibles por sus atributos, sino que es necesario que también estén accesibles por su nombre. Cualquier diccionario necesita una entrada directa a los símbolos que contiene.

Así pues, para nuestro diccionario, además de la localización por atributos, hemos introducido una entrada directa por concepto o nombre. Para ello, tenemos una tabla o lista de CDs. La cabeza de cada CD es el nombre de un concepto del diccionario. El CD de la tabla de entradas al diccionario y la descripción del concepto en la memoria están unidos por un enlace (LINK-R) rotulado como "concepto".

Esta tabla de entradas permitirá la definición de sinónimos. Serían aquellas entradas de la tabla que apuntan a la misma definición del concepto en la memoria. Cuando se introduce un concepto con distinto nombre pero con la misma descripción que otro que ya está en la memoria, el sistema lo notifica al usuario y le pregunta si son conceptos sinónimos. En caso afirmativo, el sistema sólo añade la entrada de la tabla enlazándola con la descripción ya existente en la memoria. En caso negativo, el sistema pide que distinga, mediante algún par atributo-valor, las diferencias que existen entre los dos conceptos. Esta última característica no está aún implementada en el prototipo actual.

La entrada directa por nombre no impide que en un CD o MOP de la memoria pueda existir el *slot* "nombre", u otro *slot* cualquiera, cuyo *filler* coincida con el valor de esa entrada directa por nombre. Lo que admitimos con esto, es que pueden existir atributos de los conceptos que -como "nombre"- pueden ser utilizados para representarlos o nombrarlos.

Las nociones de acciones u operaciones que se aportan al sistema son las usuales en el dominio elegido. En nuestro caso, contar, recorrer, ordenar, invertir, buscar, etc.

La información que se da de cada operacion consiste básicamente en la clase de operación a la que pertenece, así como aquellas otras operaciones más elementales en que puede descomponerse; vendría a ser por tanto, una regla muy simple de cómo solucionar problemas en los que interviene esa acción. Con clase de operación nos referimos a aquellas operaciones que sirven de base para ejecutar otras. Por ejemplo, tanto para contar el número de elementos de una lista como para localizar un elemento concreto, lo que hay que hacer es recorrer dicha lista; así, CONTAR y BUSCAR son de la clase RECORRER.

Veamos algunos ejemplos. Para poder aportar al sistema las nociones de CONTAR y BUSCAR, tendremos primero que darle la noción de RECORRER, ya que las dos anteriores descansan sobre esta última. Así pues, inicialmente, se introduce el concepto de RECORRER como una clase de operaciones, y dando las operaciones en las que se puede descomponer (una regla de producción): una inicialización en la que se apunta al primer

elemento de la estructura, y una operación sobre cada elemento de la estructura de datos que consiste en posicionarse sobre otro elemento que aún no se haya visitado. Junto con el concepto, se introduce también la entrada directa por nombre al diccionario.

El CD que se muestra a continuación sería el que está contenido en el campo CONTENTS del MOP que describe el concepto recorrer. (Siempre que hablemos de abstracciones y mostremos un CD, nos referiremos al contenido en el campo CONTENTS del MOP que representa dicha abstracción).

```
ACCION clase RECORRER
       ini-clase ACCION tipo APUNTAR
                        objeto PUNTERO
                        a PRIM-ELEMENTO de COLEC-ORGANIZADA
       cada-elem-clase ACCION tipo APUNTAR
                              objeto PUNTERO
                              a SIG-ELEMENTO de COLEC-ORGANIZADA
```

A continuación se introduce CONTAR y BUSCAR, que quedarían como una especialización de RECORRER y por tanto, heredarían la información propia de éste.

```
ACCION tipo  CONTAR
       clase RECORRER
       ini-tipo ACCION tipo INICIALIZAR
                       objeto CONTADOR
                       cantidad UNO
       cada-elem-tipo ACCION tipo INCREMENTAR
                             objeto CONTADOR
                             cantidad UNO

ACCION tipo BUSCAR
       clase RECORRER
       cada-elem-tipo ACCION tipo PREGUNTAR
                             objeto ESTADO sujeto ESTE-ELEMENTO
                                           es ELEMENTO-BUSCADO
```

En la figura 4.2 se ve cómo quedaría la memoria del diccionario. No se han puesto todos los índices por los que quedarían indexados los MOPs; sólo se ha tenido en cuenta el índice "clase" para el MOP recorrer, y el índice o *slot* "tipo" para contar y buscar.

Dentro del diccionario, también tenemos información sobre objetos relevantes en programación. Son: tipos de datos, constantes, variaién tenemos información sobre objetos relevantes en programación. Son: tipos de datos, constantes, variables, tipos abstractos de datos, parámetros, etc.

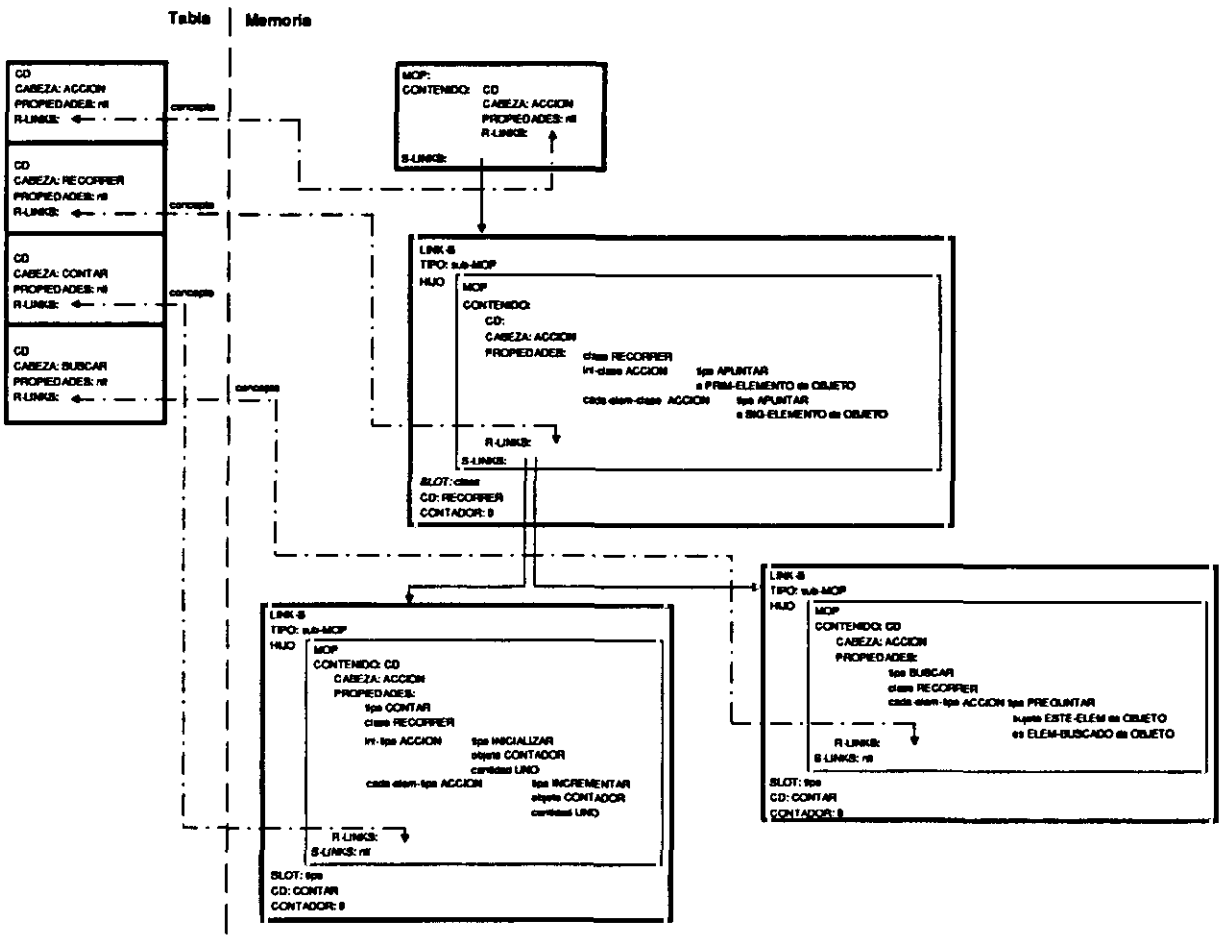

Figura 4.2

Respecto a los objetos, lo que se aporta al sistema es su definición a través de sus atributos. Bajo estas definiciones se colgarán las distintas particularizaciones, permitiendo establecer una clasificación de los objetos.

4.2.2 Biblioteca de casos. Descripción de un caso

Tal como se ha indicado, una vez proporcionado al sistema el diccionario, se introducen los casos de entrenamiento.

Cada caso consta del enunciado de un problema y una solución a dicho problema. Esta solución se describe por el método de los refinamientos sucesivos (Wirth, 1971), descomponiendo el problema en otros menores que, juntos, solucionan el anterior. A su vez, cada uno de los subproblemas se describe de la misma forma; es decir, tiene un enunciado y su solución se describe como descomposición en otros menores, y así sucesi-

vamente hasta que se llega a un problema directamente resoluble. De esta manera, se construye una jerarquía de la solución.

Esta jerarquía puede representarse como un árbol de grado N. El nodo raiz sería el enunciado del problema principal a resolver. Cada uno de los nodos hijos serían sub-problemas del anterior. Los nodos terminales serían los problemas con una solución directamente traducible al lenguaje de programación imperativo elegido.

De este modo, cada nodo del árbol es el enunciado de un problema. Cada uno de estos enunciados, se representa con un CD. Por la misma naturaleza de la descomposición de un problema en sub-problemas, el conjunto de primitivas -vocablos- que se utilizan para describir el problema principal es más amplio que en los sucesivos niveles de refinamiento. Así, a medida que bajamos por la jerarquía, pasamos a una descripción expresada con un conjunto más limitado de vocablos que normalmente se conoce como pseudocódigo, para terminar en un formalismo directamente traducible a código. Hay que tener en cuenta que todos los niveles de refinamiento comparten un mismo lenguaje de descripción y un mismo formalismo. El lenguaje es único aunque en los niveles superiores se utilicen términos más abstractos, y en los inferiores, términos más concretos.

El árbol de grado N que se forma por la descomposición de un problema en otros menores, se implementa con un árbol binario por la relación hijo-hermano. En esta implementación normalmente, por convenio, el enlace izquierdo de un nodo enlaza con su primer hijo y el enlace derecho de un nodo enlaza con su siguiente hermano.

Así, en un CD que corresponda a un nodo del árbol hay un LINK-R - rotulado como "refinamiento" (enlace izquierdo)- que apunta al primer CD subproblema en que se descompone. Este CD subproblema tendrá otro LINK-R -rotulado como "siguiente" (enlace derecho)- que apuntará a su hermano (siguiente subproblema en el mismo nivel de la jerarquía).

Dentro de las operaciones que se realizan en los programas, y a cualquier nivel de refinamiento, vamos a considerar dos tipos fundamentales: las que consisten en ejecutar una serie de acciones (aquellas que en el último nivel de refinamiento -código-, llamaríamos "sentencias") y las que consisten en determinar el estado de determinados objetos ("condiciones" en un nivel de código).

En adelante, para facilitar las cosas, hablaremos de sentencias y condiciones aunque estos nombres sólo se podrían utilizar estrictamente en el último nivel de refinamiento, directamente traducible a código.

Como en cualquier clasificación clásica (Seymour, 1987), en nuestro sistema tenemos tres tipos de sentencias según el flujo de control que marcan en el programa: secuencial, condicional o selectivo y repetitivo.

Aunque en la mayoría de los lenguajes de programación existen diveros tipos de bucles ("for", "while", "repeat"), y varias sentencias selectivas ("if", "case") nosotros sólo hemos incluido un bucle ("while") y una selección ("if"). El motivo está en que ya en 1966, Böhm (1966) demostró que bastan tres estructuras de control (secuencia, selección e iteración) para expresar cualquier flujo de control de un programa representable en un organigrama. El posprocesador, en caso necesario, se encargaría de trasformar los bucles y selecciones a los más convenientes. Para los casos de entrenamiento, sería el preprocesador el encargado de transformar los distintos tipos de bucles y selecciones a los que consideramos aquí.

También siguiendo manuales clásicos (Wirth, 1973), trataremos las sentencias selectivas e iterativas como un bloque (es decir, un enunciado repetitivo o selectivo es un nodo del árbol; a su vez, este nodo es un subárbol que tiene como hijo la condición por la que se pregunta y las sentencias que hay que ejecutar en su caso). Simultáneamente, de acuerdo con lo que es propio de la programación imperativa, en nuestro sistema se entiende que hay un flujo secuencial si no se dice lo contrario.

El otro elemento a tener en cuenta en un programa son las condiciones. Al igual que las sentencias, son un nodo en el árbol de refinamientos, y por tanto se representan con CDs.

También las condiciones se pueden descomponer en otras más sencillas enlazadas entre sí por conectivos lógicos. En estos casos el CD padre tiene un LINK-R rotulado "refinamiento" al primer CD que representa la primera sub-condición por la que preguntar. De este CD tendremos un LINK-R rotulado con el nombre de la conectiva lógica correspondiente ("y-logico", "o-logico", etc.) a su hermano (la siguiente condición a testear), y así sucesivamente.

Si tengo varios niveles de refinamiento en las condiciones, primero se operan entre sí todas las condiciones de un mismo nivel (como si estuvieran encerradas por paréntesis). Dentro de un mismo nivel, como orden de precedencia se toma el convenido normalmente por defecto (los "o" lógicos marcan la separación). Si se deseara otra agrupación, se tendría que recurrir a otro nivel de refinamiento.

Como ya se ha apuntado, las sentencias que no son secuenciales, se tratan como un bloque, un nodo en el árbol. Cada uno de estos nodos es a su vez la raíz de un sub-árbol donde: el nodo raíz es simplemente un CD que indica el tipo de flujo de que se trata

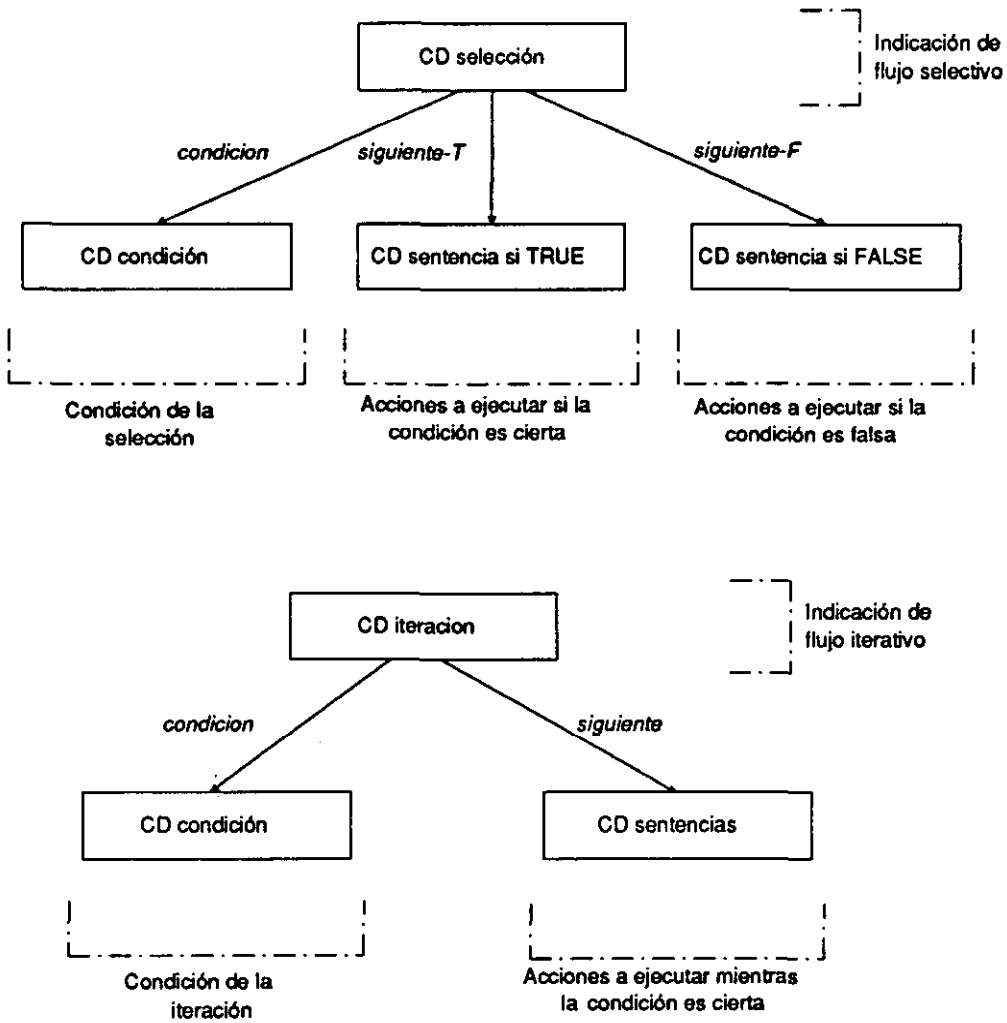

Figura 4.3

(selectivo o iterativo). Como nodos hijos tiene la condición y las sentencias a ejecutar en su caso. Este subárbol se implementa de nuevo por la relación hijo-hermano.

Así pues tenemos:

- Para una sentencia selectiva, el CD nodo raíz tiene dos o tres (según los casos) LINKs-R etiquetados como "condición", "siguiente-T" (siguiente si cierto) y "siguiente-F" (siguiente si falso) a los CDs correspondientes (ver figura 4.3 (a). Cada uno de estos CDs, podría descomponerse en otros.

- Para una sentencia repetitiva, el CD nodo raíz tiene dos LINKs-R a CDs etiquetados como "condicion" y "contiene" (ver figura 4.3 (b). También estos últimos CDs pueden descomponerse en otros.

Para resumir, representando en letra negrita las etiquetas de los LINKs-R, el diagrama BNF que obtendríamos para representar las soluciones, sería el siguiente:

```
<programa> ::= <lista sentencias>
<lista sentencias> ::= <sentencia> {siguiente <sentencia>}
<sentencia> ::= <sentencia secuencial> | <sentencia no secuencial>
<sentencia secuencial> ::= <CD secuencial> |
                           <CD secuencial> refinamiento <lista sentencias>
<sentencia no secuencial> ::= <sentencia selectiva> |
                              <sentencia repetitiva>
<sentencia selectiva> ::= <CD selectivo> condicion <lista condiciones>
                                         siguiente-T <lista sentencias> |
                          <CD selectivo> condicion <lista condiciones>
                                         siguiente-T <lista sentencias>
                                         siguiente-F <lista sentencias>
<sentencia repetitiva> ::= <CD repetitivo> condicion <lista condiciones>
                                           contiene <lista sentencias>

<lista condiciones> ::= <condicion> |
                        <condicion> y-logico <lista condiciones> |
                        <condicion> o-logico <lista condiciones>
<condicion> ::= <CD condicion> |
                <CD condicion> refinamiento <lista condiciones>
```

Algunos de los CDs que constituyen el enunciado de un problema o subproblema, tendrán precondiciones y postcondiciones que deben verificarse para que la solución global sea correcta. Estas pre y poscondiciones se indicarán con los *slots* "precondicion" y "poscondicion" respectivamente. Si es *true* se omite el *slot*. La utilidad de estas precondiciones se describirá al hablar de los métodos de inferencia.

Un ejemplo de precondición sería el declarar que antes de utilizar el valor de una variable, ésta debe ser inicializada, es decir, su contenido no es "basura". Por ejemplo:

```
ACCION tipo ASIGNAR
       a CONTADOR
       objeto ACCION tipo SUMAR
                     objeto1 CONTADOR
                     objeto2 UNO
      precondicion ESTADO sujeto CONTADOR
                          noes BASURA
```

A continuación describiremos con detalle un caso concreto.

Un caso o programa completo se describe creando todos los CDs que componen la solución (todos los nodos del árbol) con la función 'list- >cd' y relacionándolos con los

enlaces adecuados con la función 'crea-link-r' formando de esta manera los distintos niveles de refinamiento.

Supongamos como ejemplo, "Contar el número de nodos de una lista enlazada simple". Una posible solución al problema sería descomponerlo en dos, cuando la lista es vacía y cuando no lo es. En el primer caso, el problema está resuelto, el resultado es cero. En el segundo caso, el problema se resuelve realizando los siguientes pasos: inicializar el contador a uno; establecer un puntero auxiliar apuntando al primer elemento de la lista; entrar en un bucle: mientras no termine la lista, incrementar el contador en uno y avanzar el puntero al siguiente elemento de la lista.

Todo este proceso, en forma de árbol y representando enunciado y solución, quedaría como queda reflejado en la figura 4.4. Con los puntos queremos simbolizar que son nodos hijos, aunque el enlace se establezca por la relación hijo-hermano.

La implementación en CDs de cada uno de estos nodos quedaría como sigue.

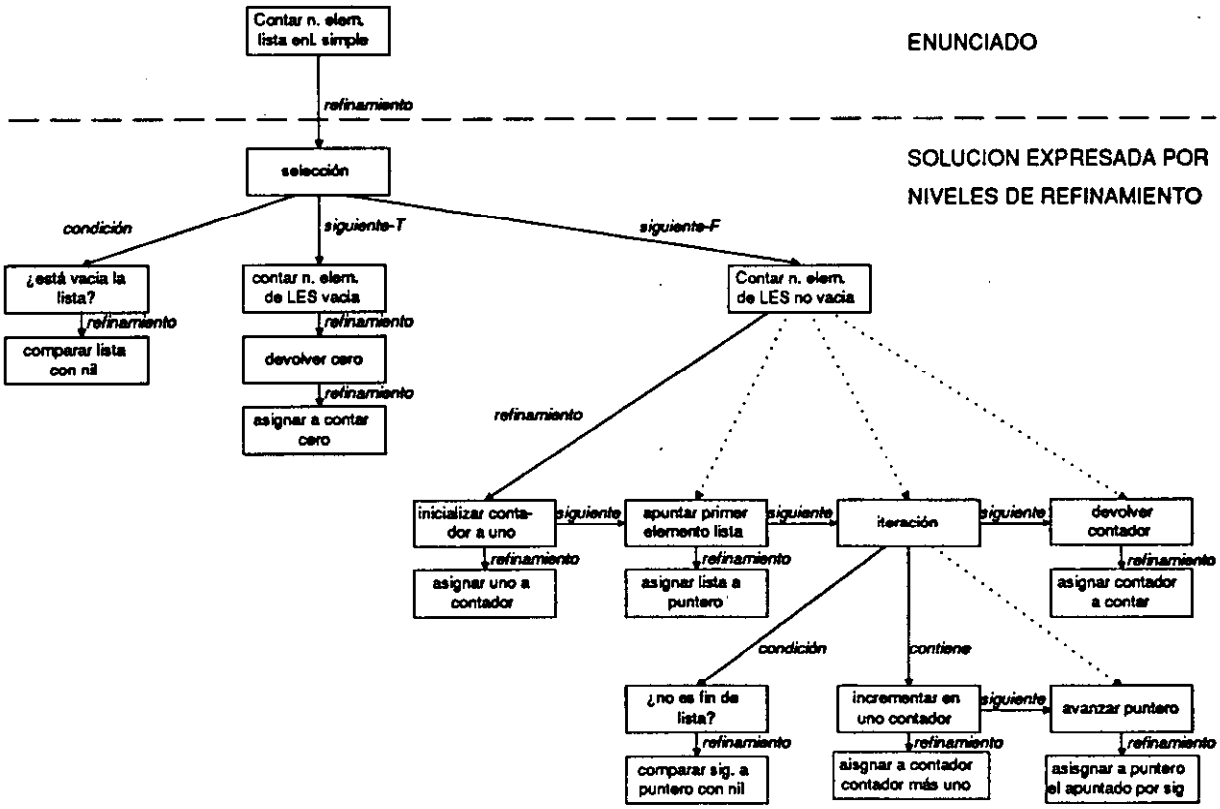

Figura 4.4

El nodo de la raiz sería el enunciado del problema:

```
(setq C1-E (list->cd
              '(ACCION ((tipo CONTAR)
                        (objeto NUM-ELEMENTOS
                              ((de (OBJETO ((esun COLECCION-ORGANIZADA)
                                            (num-anteriores UNO)
                                            (num-siguientes UNO)
                                            (implementada ENLAZADA)
                                            (forma SIMPLE)
                                            (longitud INDETERMINADA)))
                                   )))
)))
```

Anteriormente hemos dicho que las sentencias selectivas o iterativas se consideraban como un único bloque. Así, en este caso, el nodo raíz tiene un único nodo hijo que consiste en determinar la longitud de la lista, que permite dividir el problema en dos: cuando la lista está vacia, y cuando no lo está. Así pues, tal como se ve en el gráfico anterior, este nodo tendrá tres nodos hijos: la condición, la acción a realizar en caso de cumplirse la condición, y las sentencias a ejecutar en caso de no verificarse.

En forma de CDs esta sentencia selectiva sería:

```
; - Indicación de sentencia selectiva -

(setq C1-A (list->cd '(ACCION ((tipo SELECTIVA)))))

; - Condición por la que se pregunta -

(setq C1-AC (list->cd
               '(ACCION ((tipo PREGUNTAR)
                         (objeto ESTADO
                             ((sujeto LONGITUD
                                ((de (OBJETO ((esun COLECCION-ORGANIZADA)
                                              (num-anteriores UNO)
                                              (num-siguientes UNO)
                                              (implementada ENLAZADA)
                                              (forma SIMPLE)))))
                                 (es NULA)))))
))
```

```
; - Acción a realizar si la condición es cierta -
(setq C1-AT (list->cd
              '(ACCION ((tipo CONTAR)
                        (objeto NUM-ELEMENTOS
                              ((de (OBJETO ((esun COLECCION-ORGANIZADA)
                                            (num-anteriores UNO)
                                            (num-siguientes UNO)
                                            (implementada ENLAZADA)
                                            (forma SIMPLE)
                                            (longitud NULA)))
                                   )))
)))

; - Accion a realizar si la condición es falsa -

(setq C1-AF (list->cd
              '(ACCION ((tipo CONTAR)
                        (objeto NUM-ELEMENTOS
                              ((de (OBJETO ((esun COLECCION-ORGANIZADA)
                                            (num-anteriores UNO)
                                            (num-siguientes UNO)
                                            (implementada ENLAZADA)
                                            (forma SIMPLE)
                                            (longitud NO-NULA)))
                                   )))
)))
```

Estos cuatro CDs se deben relacionar con los enlaces LINK-R tal como se expresa en las siguientes llamadas a la función 'crea-r-link':

```
(crea-link-r 'refinamiento C1-E C1-A)        ; Nivel de refinamiento

(crea-link-r 'condicion   C1-A C1-AC)
(crea-link-r 'siguiente-T C1-A C1-AT)
(crea-link-r 'siguiente-F C1-A C1-AF)
```

Nos centramos ahora, dejando el resto para no alargarnos, en el nodo concreto que indica "contar el número de elementos de una lista enlazada simple no vacía". Este problema se descompone en inicializar un contador a uno, inicializar un puntero apuntando a lista, recorrer la lista aumentando el contador por cada nodo de la lista -una sentencia repetitiva- y por último, devolver el contador.

Su implementación en CDs sería:

```
; - Inicializar el contador a uno -

(setq C1-AFr1    (list->cd '(ACCION ((tipo INICIALIZAR)
                                     (objeto CONTADOR)
                                     (cantidad UNO)))))

; - Inicializar un puntero al primer elemento de la lista -

(setq C1-AFr2 (list->cd
               '(ACCION ((tipo APUNTAR)
                         (objeto PUNTERO)
                         (a PRIM-ELEMENTO
                                ((de (OBJETO ((esun COLECCION-ORGANIZADA)
                                              (num-anteriores UNO)
                                              (num-siguientes UNO)
                                              (implementada ENLAZADA)
                                              (forma SIMPLE))))
                          ))
))))

; - Recorrer la lista incrementando el contador -

; Indicación de sentencia repetitiva

(setq C1-AFr3  (list->cd '(ACTION ((tipo REPETITIVA)))))

; Condición por la que se pregunta

(setq C1-AFr3C (list->cd
               '(ACCION ((tipo PREGUNTAR)
                         (objeto ESTADO
                             ((sujeto ESTE-ELEMENTO
                                ((de (OBJETO ((esun COLECCION-ORGANIZADA)
                                              (num-anteriores UNO)
                                              (num-siguientes UNO)
                                              (implementada ENLAZADA)
                                              (forma SIMPLE)))))
                              (noes ULTIMO-ELEMENTO
                                ((de (OBJETO ((esun COLECCION-ORGANIZADA)
                                              (num-anteriores UNO)
                                              (num-siguientes UNO)
                                              (implementada ENLAZADA)
                                              (forma SIMPLE)))))
               )))))
))
```

```
; Acciones a realizar mientras se cumpla la condición

; Incrementar el contador
(setq C1-AFr31 (list->cd '(ACTION ((tipo INCREMENTAR)
                                   (objeto CONTADOR)
                                   (cantidad UNO)))))

; Avanzar el puntero
(setq C1-AFr32 (list->cd
                '(ACTION ((tipo APUNTAR)
                          (objeto PUNTERO)
                          (a SIG-ELEMENTO
                               ((de (OBJETO ((esun COLECCION-ORGANIZADA)
                                             (num-anteriores UNO)
                                             (num-siguientes UNO)
                                             (implementada ENLAZADA)
                                             (forma SIMPLE)))))
                 )))
))

; - Devolver el contador -

(setq C1-AFr4   (list->cd '(ACCION ((tipo DEVOLVER)
                                    (objeto CONTADOR)))))
```

Sus relaciones:

```
(crea-link-r 'refinamiento C1-AF C1-AFr1) ; Primer nodo en el refinamiento
(crea-link-r 'siguiente C1-AFr1 C1-AFr2)  ; Hermano del anterior
(crea-link-r 'siguiente C1-AFr2 C1-AFr3)  ; Hermano del anterior
(crea-link-r 'siguiente C1-AFr3 C1-AFr4)  ; Hermano del anterior

(crea-link-r 'condicion C1-AFr3  C1-AFr3C) ; Condición de la iteración
(crea-link-r 'contiene  C1-AFr3  C1-AFr31) ; Sentencias a ejecutar
(crea-link-r 'siguiente C1-Afr31 C1-AFr32) ;
```

Cada uno de estos nodos se refina. Por ejemplo, inicializar el contador a uno sería:

```
(setq C1-AFr1r  (list->cd '(ACCION  ((tipo ASIGNAR)
                                     (a CONTADOR)
                                     (objeto 1)))))
```

que ya es un problema directamente traducible a un lenguaje imperativo. La relación con su padre se establecería con:

```
(crea-link-r 'refinamiento C1-AFr1  C1-AFr1r)
```

4.3 INDEXACION EN MEMORIA DE LA INFORMACION SUMINISTRADA AL SISTEMA

A continuación vamos a ver los procesos que incorporan información al sistema, en nuestro prototipo en particular, y las consecuencias que estas operaciones tienen sobre la memoria.

En nuestro sistema tenemos dos tipos de información de entrada: los conceptos del diccionario y los casos de entrenamiento del sistema.

En el capítulo anterior se ha descrito cómo se indexan CDs simples y redes de CDs en la memoria del sistema del que partimos. En este capítulo describiremos cómo indexar el conocimiento propio de nuestro sistema: el diccionario de conceptos y los casos de entrenamiento. Los primeros son CDs y MOPs, y los segundos redes de CDs.

4.3.1 Indexación de los conceptos del diccionario

La descripción de los conceptos del diccionario, como ya se ha expuesto, se hace por medio de CDs y MOPs. La forma de indexar CDs se explicó en el capítulo anterior. Para indexar nuestros CDs del diccionario, lo único que hay que añadir es la entrada directa. También hay pequeñas variaciones cuando el concepto se representa en un MOP.

Los procesos a realizar son los siguientes:

a) Se crea un CD con la función 'list-> cd'. Este CD contiene los atributos del concepto que se está definiendo. Si el concepto se va a representar con un MOP, este CD es el que irá en el campo CONTENTS de ese MOP.

b) Se crea el CD que se utilizará para la entrada directa al diccionario.

c) Se relacionan los dos CDs con la función 'crea-r-link'. El LINK-R que se crea se rotula como "concepto". Con esto lo que hacemos es establecer la entrada directa para el diccionario.

d) Si el concepto se va a representar con un MOP, se crea un MOP. En su campo CONTENTS se pone el CD creado en el paso a); el campo S-LINKS se deja a nil -por ahora no tiene especializaciones-;

e) Se indexa el concepto: CD o MOP según sea el caso. Los CDs se indexan tal como se indicó en el capítulo anterior. Para indexar un MOP se buscan todos aquellos MOPs de la memoria que sean terminales y compatibles con el de la entrada. Decimos que dos MOPs son compatibles si lo son los CDs respectivos contenidos en el campo CONTENTS. Se

crean los índices LINK-S que permitirán referenciar el nuevo MOP desde los MOPs compatibles que han sido localizados. Por último -y como en el caso de los CDs- si es posible, se intenta generalizar el contenido de la nueva información introducida y de otras similares introducidas en operaciones anteriores. Estos nuevos sub-MOPs -si existen- se incorporan a la memoria indexándolos en la red y re-indexando los MOPs de entrada bajo estos nuevos sub-MOPs. El sistema, cada vez que construye un MOP abstracto pregunta al usuario si esa abstracción corresponde a un concepto conocido y quiere darle un nombre. Si el usuario proporciona un nombre, éste se introduce en la tabla de entrada directa al diccionario.

Tomamos como ejemplo la lista. Por lista entendemos una colección de elementos organizada, donde a cada elemento le precede un elemento y le sigue otro (Seymour, 1987) (como excepciones tendríamos el primer y último elementos, pero no los vamos a considerar en este ejemplo). Hasta aquí tendríamos la definición más general de lista. Existen algunas caractr en este ejemplo). Hasta aquí tendríamos la definición más general de lista. Existen algunas características que nos permiten diferenciar distintos tipos de listas. Por ejemplo, el orden de precedencia entre los distintos elementos se determina o bien secuencialmente (el siguiente elemento está en la siguiente posición), o bien de forma no secuencial, dando de forma explícita la posición del siguiente elemento. Como ejemplos de estos dos casos tendríamos las listas secuenciales o las listas enlazadas, entre otras posibilidades.

Tenemos por tanto dos posibles tipos de listas que hemos denominado: "Lista Enlazada" y "Lista Secuencial". Su representación en CDs sería la siguiente:

```
OBJETO esun COLECCION_ORGANIZADA
       num-anteriores UNO
       num-siguientes UNO
       implementada ENLAZADA

OBJETO esun COLECCION_ORGANIZADA
       num-anteriores UNO
       num-siguientes UNO
       implementada SECUENCIAL
```

Estos CDs se introducen en el diccionario de conceptos aportando su nombre: "Lista Enlazada" y "Lista Secuencial" respectivamente. La memoria quedaría tal como se describe en la figura 4.5. Para simplificar sólo hemos indexado los CDs por los *slots* "esun" e "implementada".

Al organizar toda esta información en la memoria, el sistema encuentra atributos comunes y obtiene abstracciones. Cada una de estas abstracciones se muestra al usuario y se le pregunta si esa abstracción corresponde a algún concepto conocido. En caso afirma-

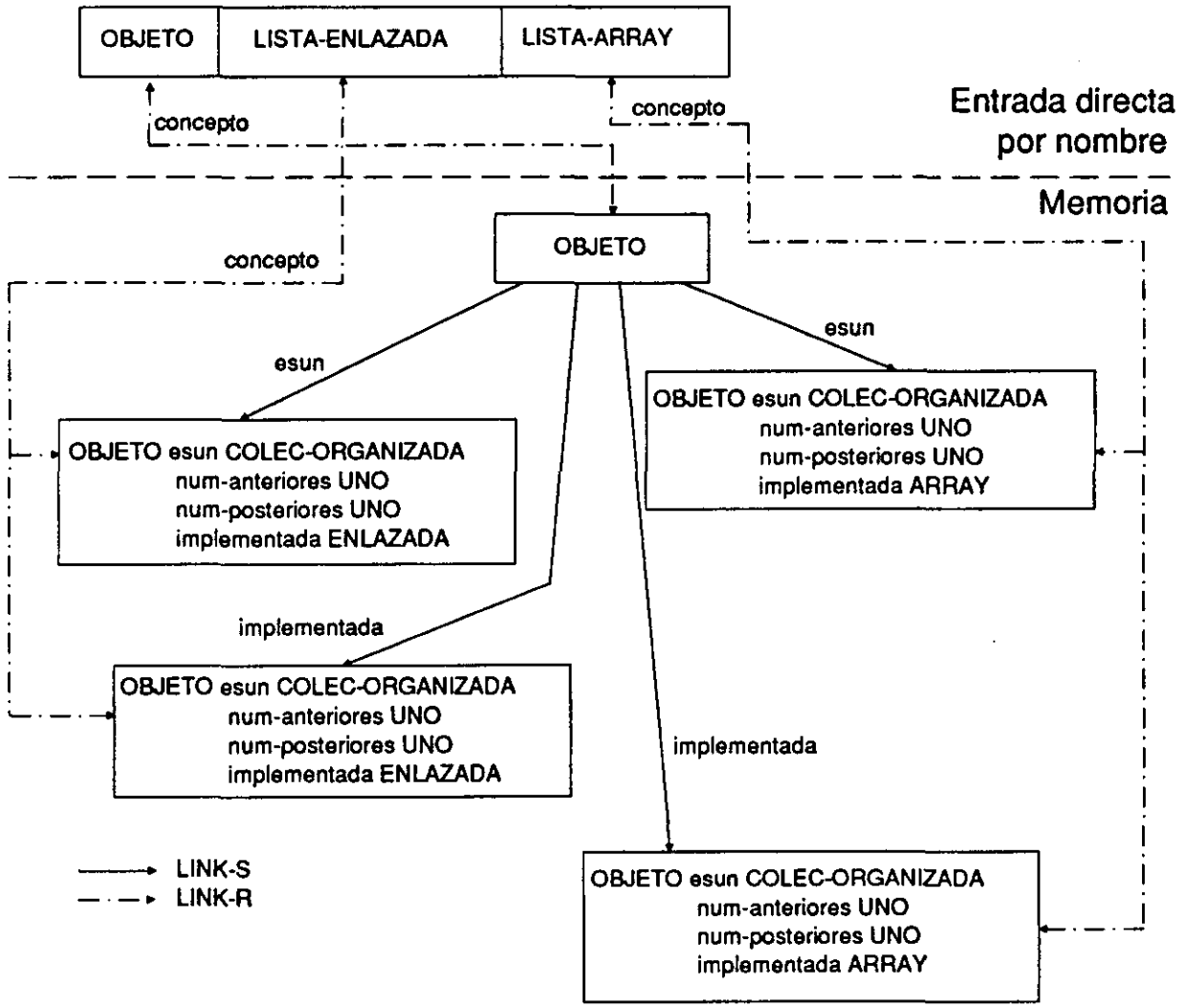

Figura 4.5

tivo, el usuario introduce el nombre ese concepto y por tanto la abstracción tendrá una entrada directa en el diccionario. Si no corresponde a ningún concepto conocido, abstracción tendrá una entrada directa en el diccionario. Si no corresponde a ningún concepto conocido, la abstracción se hace igualmente, pero en el diccionario no consta una entrada directa.

En el ejemplo anterior, el sistema crea una abstracción:

```
OBJETO esun COLECCION_ORGANIZADA
       num-anteriores UNO
       num-siguientes UNO
```

El sistema pregunta al usuario si esta abstracción corresponde con algún concepto conocido. El usuario responde afirmativamente y le dice que esa abstracción corresponde al concepto de "Lista". La memoria queda tal como se describe en la figura 4.6

4.3.2 Indexación de casos de entrenamiento

Un caso es una red de CDs cuyos enlaces indican la secuencialidad del programa.

El usuario introduce estos casos de entrenamiento con los siguientes pasos:

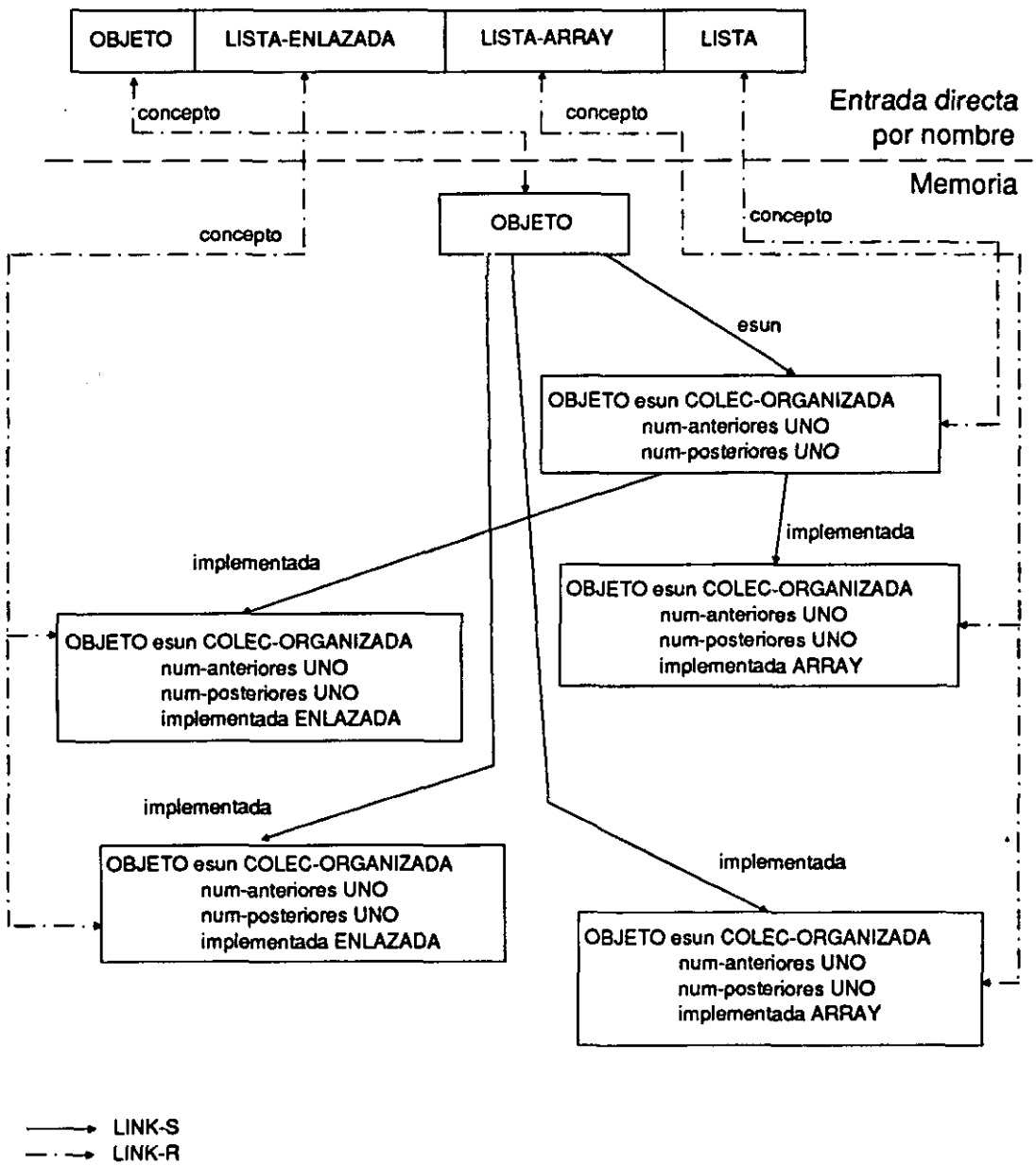

Figura 4.6

Primero, con las funciones 'list->cd' y 'crea-link-r', establece la red de CDs que forma el enunciado con su solución en forma de refinamientos sucesivos.

Actualmente, este proceso hay que realizarlo a mano. Como ya se apuntó, no disponemos de un *parser*, incorporado en el sistema, que sería el encargado de construir estos CDs y sus enlaces a partir del texto, utilizando la propia memoria como fuente de información contextual.

A continuación, ejecuta la función 'indexa-caso' a la que se le pasa como parámetro el CD que representa el problema principal.

Inicialmente, el sistema indexa todo el árbol de solución. Es decir, indexa cada uno de los CDs que forman el enunciado y su solución, tal como se ha indicado al hablar de la indexación de CDs simples.

Estos CDs están vistos desde el púnto de vista de la acción, es decir, su cabecera es siempre ACCION, por ejemplo:

```
ACCION tipo CONTAR
       objeto NUM-ELEMENTOS de OBJETO esun COLECCION-ORGANIZADA
                                      num-anteriores UNO
                                      num-siguientes UNO
                                      implementada ENLAZADA
                                      forma SIMPLE
```

Algunos de estos CDs trabajan sobre una estructura de datos, es decir, la operación que realizan tienen que ver con alguna estructura de datos como sucede en el ejemplo anterior.

Para estas acciones, en las que está involucrada una estructura de datos, el sistema establece un enlace (LINK-R) entre la acción -una vez la ha indexado en la memoria- y la estructura de datos -también indexada en la memoria-. El LINK-R se rotula con el mismo nombre de la acción. De esta forma, tengo enlaces directos a todas aquellas operaciones en la que se trabaja sobre una estructura de datos concreta; en el ejemplo que estamos tratando, todas aquellas operaciones relacionadas con la lista enlazada simple.

Veamos cómo quedaría parte de la memoria después de indexar el programa "Contar el número de elementos de una lista enlazada simple". Para que quede más claro, sólo se dibujarán los bloques del primer nivel de refinamiento del problema original y el primero de uno de los subproblemas.

En la figura 4.7, se describe la solución que divide el problema en dos: cuando la lista es vacía y cuando no lo es. Es decir, el enunciado y su primer nivel de refinamiento.

En la figura 4.8, se describe el problema de contar el número de elementos de una lista enlazada simple no vacía; el primer nivel de refinamiento para uno de los subproblemas de los dos en que ha quedado dividido el el problema anterior..

Sólo se han puesto algunos índices por los que estarían indexados los CDs. También se han omitido algunos de los MOPs intermedios en la jerarquía de abstracción entre el MOP ACCION y los CDs terminales. Puede verse cómo aquellas acciones que se realizan sobre una estructura de datos, están enlazadas con la definición de esa estructura de datos que existe en el diccionario de conceptos.

4.4 INFERENCIA DE LA SOLUCION DE NUEVOS CASOS. ANALOGIA.

A continuación se describirán los procesos que realizará el sistema para obtener la solución de un nuevo caso que se le plantee.

Ante la petición de la solución de un problema, el sistema busca -en la memoria- aquel o aquellos casos más similares al nuevo problema. Dependiendo del *matching* (emparejamiento) que pueda establecer entre el caso de la entrada y los recuperados de la memoria, obtendrá una posible solución, o bien, notificará al usuario que no tiene información suficiente para generar dicha solución. El usuario, si lo desea, puede introducir la solución, aumentando la biblioteca de casos.

Antes de detallar los tipos de *matching* que el sistema puede encontrar y los procesos correspondientes -según el emparejamiento- para obtener la solución, caracterizaremos los problemas y la forma en que el usuario solicita la solución de un nuevo problema.

4.4.1 Caracterización de un problema

Dentro de cada problema, podemos distinguir dos partes fundamentales: el objeto que se va a utilizar y la operación o función que se va a ejecutar sobre el mismo.

En nuestro sistema estos datos vienen dados por *fillers* del CD que plantea el problema. El *slot* "tipo" indica de qué operación se trata; "de" (complemento de nombre del caso gramatical complemento directo) proporciona el objeto sobre el que se realiza la operación. Por otro lado, los enunciados se enfocan siempre desde el punto de vista de la acción, de forma que la cabecera será siempre ACCION.

Para representar el enunciado: "*CONTAR* el número de elementos de una *LISTA ENLAZADA SIMPLE*", de la que se supone se desconoce la longitud de la misma, tendríamos el siguiente CD:

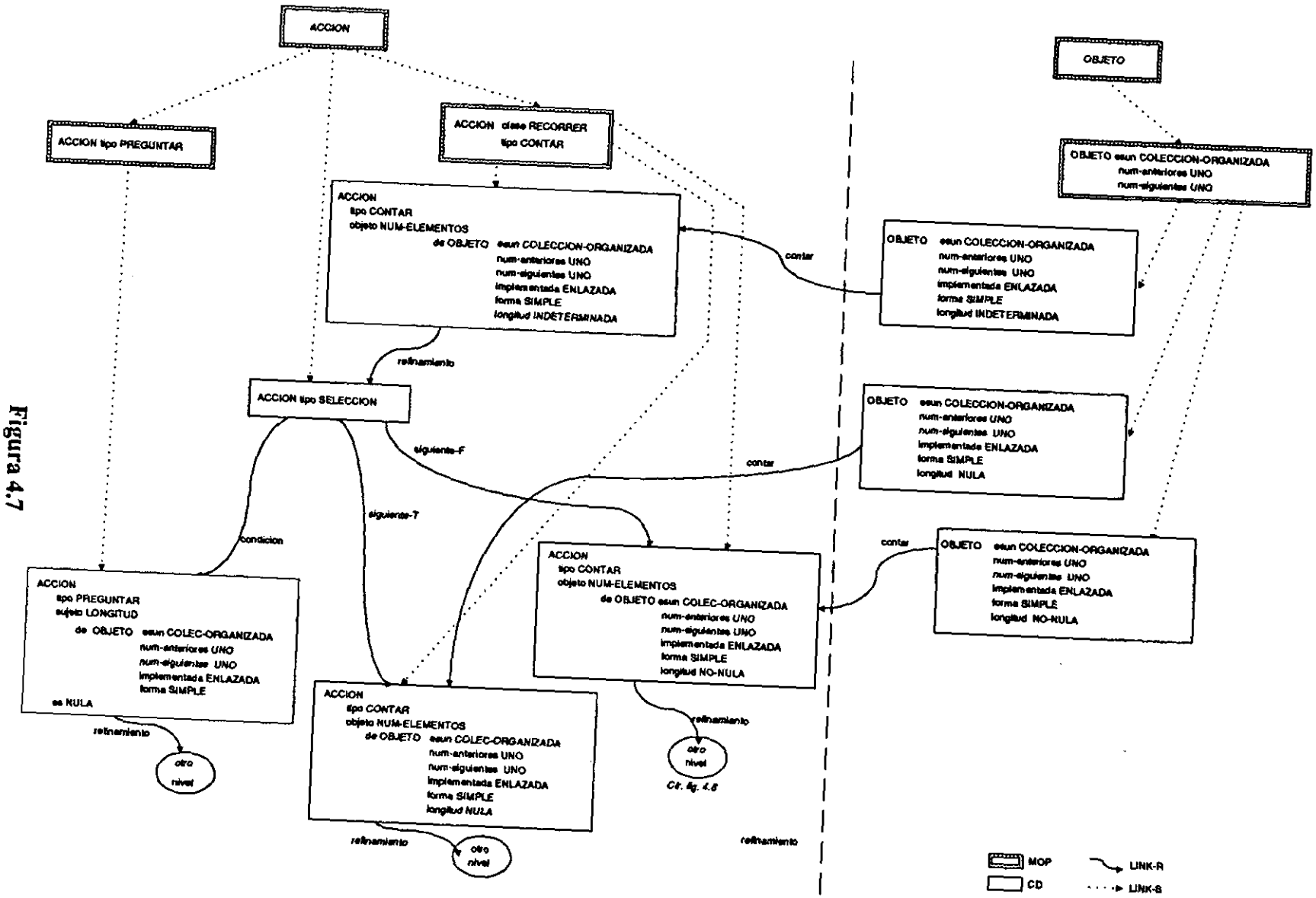
ACCION
ACCION tipo PREGUNTAR
ACCION clase RECORRER
tipo CONTAR
ACCION
tipo CONTAR
objeto NUM-ELEMENTOS
de OBJETO esun COLECCION-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
implementada ENLAZADA
forma SIMPLE
longitud INDETERMINADA
contar
refinamiento
ACCION tipo SELECCION
siguiente-F
siguiente-T
condicion
contar
ACCION
tipo PREGUNTAR
sujeto LONGITUD
de OBJETO esun COLEC-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
implementada ENLAZADA
forma SIMPLE
es NULA
refinamiento
otro nivel
ACCION
tipo CONTAR
objeto NUM-ELEMENTOS
de OBJETO esun COLEC-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
implementada ENLAZADA
forma SIMPLE
longitud NULA
refinamiento
otro nivel
ACCION
tipo CONTAR
objeto NUM-ELEMENTOS
de OBJETO esun COLEC-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
implementada ENLAZADA
forma SIMPLE
longitud NO-NULA
refinamiento
otro nivel
Cf. fig. 4.8
OBJETO
OBJETO esun COLECCION-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
OBJETO esun COLECCION-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
implementada ENLAZADA
forma SIMPLE
longitud INDETERMINADA
OBJETO esun COLECCION-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
implementada ENLAZADA
forma SIMPLE
longitud NULA
contar
OBJETO esun COLECCION-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
implementada ENLAZADA
forma SIMPLE
longitud NO-NULA
contar
MOP
CD
LINK-R
LINK-S

Figura 4.7

```
ACCION tipo CONTAR
       objeto NUM-ELEMENTOS de OBJETO esun COLECCION-ORGANIZADA
                                          num-anteriores UNO
                                          num-siguientes UNO
                                          implementada ENLAZADA
                                          forma SIMPLE
                                          longitud INDETERMINADA
```

Este ejemplo es simple. Podría darse el caso de que sólo se contasen determinados elementos, aquellos que cumplieran una condición dada. Estas ligaduras se especificarían en el *filler* del *slot* "objeto" y modificarían el algoritmo de resolución.

El prototipo actual no resuelve problemas en los que se entremezclan distintas operaciones con varias estructuras de datos simultáneamente. Se trata siempre de una única operación sobre una única estructura de datos.

4.4.2 Métodos de resolución

El modo de comunicación del usuario con el sistema se hace a través de la función 'problem-solver', a la que se le pasa como parámetro el CD que representa el problema a resolver. Dicho CD se ha construido a su vez con la función 'list->cd'.

Cuando al sistema se le pide que resuelva un problema, lo primero que hace es buscar en su memoria de casos los más similares, tratando de establecer analogías.

En nuestro prototipo, esta analogía es superficial, ya que busca semejanzas entre operaciones y estructuras de datos, basándose en el enunciado (lo que se solicita) y en el diccionario de conceptos. Es decir, con esta analogía, lo que se pretende es estimar la 'proximidad' de dos problemas en el dominio de la implementación, basándose en su 'proximidad' en el dominio de la especificación.

Se podría haber optado por intentar detectar analogías entre problemas que se han resuelto por un mismo método, como por ejemplo el de "divide y vencerás". Para esto sería necesario clasificar los problemas de otra forma, por el tipo de método por el que pueden ser resueltos. Simultáneamente, sería necesario aportar al sistema mucha más información -e información bastante compleja- sobre las operaciones y su 'funcionamiento'.

Una de las ventajas que se le atribuyen al CBR es que no requiere un modelo causal ni un conocimiento profundo del dominio, aunque lógicamente las dos cosas proporcionarían un mejor funcionamiento (DARPA, 1989). Por este motivo, como una primera aproximación, hemos preferido buscar y trabajar con similitudes superficiales para evitar la necesidad de introducir grandes cantidades de conocimiento al sistema, y aprovechar así, las ventajas del CBR.

Como ya se ha explicado anteriormente las operaciones están agrupadas por clases en el diccionario de conceptos (se pueden determinar por el *slot* "clase"). De forma similar, los objetos están también clasificados de forma jerárquica según los tipos de objetos (lista, árbol, etc.). Entre ellos se diferencian por su implementación, grado, etc.

Siguiendo a Miriyala & Harandi (1991), hemos establecido dos tipos de analogías: directas y aproximadas. Para establecer el tipo de analogía que existe entre dos problemas, previamente establecemos las analogías existentes entre las operaciones y las estructuras de datos, por separado.

Decimos que existe analogía directa en la operación cuando se trata de la misma operación, y que existe analogía aproximada cuando se trata de operaciones distintas pero dentro de la misma clase de operaciones. De forma similar, decimos que existe analogía directa en el objeto cuando se trata del mismo objeto; y que hay analogía sólo aproximada cuando son estructuras del mismo grupo de objetos.

Ya hemos dicho que cuando se le pide al sistema que resuelva un nuevo programa, lo primero que intenta es encontrar casos lo más similares posible.

El sistema busca casos que cumplan las siguientes analogías y por este orden:

- Analogía directa en la estructura de datos y en la operación: se trata del mismo caso, por lo que el problema ya está resuelto.

- Analogía aproximada en la estructura de datos y directa en la operación.

- Analogía directa en la estructura de datos y aproximada en la operación.

- Analogía aproximada en la estructura de datos y en la operación.

Los tres últimos tipos podríamos reagruparlos en dos: analogía directa en la operación y analogía aproximada en la operación.

Si no se encontrase en la memoria ningún caso que cumpliera alguna de estas analogías, el sistema avisaría de que no puede resolverlo. No hay que olvidar que nuestro sistema utiliza el método de razonamiento basado en casos, a través de analogías. Si no puede establecer estas analogías, la resolución cae fuera de su ámbito y no puede llevarse a cabo. Se supone que este sistema formaría parte de un sistema global que intentaría resolver el problema utilizando otros mecanismos.

El sistema inicia la resolución de un nuevo caso determinando el tipo de analogía que puede conseguir: directa o aproximada. Para eso sólo necesita detectar si existe al menos un caso en la memoria con analogía directa con el caso de la entrada. Una vez determinado

el tipo de analogía que se puede establecer, el sistema ejecuta el método de resolución correspondiente.

Detallamos a continuación cómo funciona el sistema frente a las dos situaciones posibles en que se puede encontrar: tener una analogía directa en la operación, o tenerla sólo aproximada.

Analogía Directa

Aunque basta encontrar un caso para determinar la analogía directa, el sistema busca todo el conjunto de casos de la biblioteca que tengan una analogía directa en la operación con el caso a resolver.

De este primer grupo inicial de casos selecionados, selecciona aquél que tenga una analogía directa en el objeto. Si lo encuentra, toma ese caso como el que mejor empareja, y el problema está resuelto al haber una analogía directa en la operación y en el objeto. Se trataría del caso trivial.

Si no lo encuentra, de ese primer grupo inicial, selecciona los que tengan una analogía aproximada en el objeto. De entre éstos, el que mejor empareje será el que esté más "próximo" en la clasificación de objetos dada por el diccionario de conceptos. La "proxidad" (o "distancia" mínima) depende de las relaciones establecidas en la jerarquía del diccionario de conceptos; se trata del mínimo número de "ramas" por las que hay que pasar para ir de un punto a otro de la jerarquía.

A grandes rasgos, los pasos que el sistema realiza después de obtener el caso que mejor empareja y determinar que se trata de una analogía directa en la operación, son:

Primero determina las características que diferencian el caso que se desea obtener del caso recuperado de la biblioteca. Entre estas características se encontrará el objeto, ya que sólo hemos podido establecer una analogía aproximada.

A continuación -utilizando la descomposición *top-down* de nuestro sistema, y comenzando por el nivel más abstracto- toma cada uno de los subproblemas en los que se descompone el caso recuperado. Para cada subproblema, el sistema debe resolver uno semejante; semejante porque implementará la misma operación, pero las características que no coinciden con el caso que se pretende resolver, se habrán cambiado por las nuevas. Es decir, cada uno de los subproblemas se trata como un caso de entrada que implementa la misma operación, pero donde su enunciado se ha modificando para ajustarse al nuevo objeto y características del problema propuesto original. Esta es la razón de por qué las soluciones de los subproblemas de los casos -tanto de entrenamiento como los resueltos por el propio sistema- se indexan también en la base de conocimiento bajo las operaciones que implementan.

Una vez resueltos cada uno de los subproblemas, el problema estará resuelto con sólo enlazar las soluciones de los subproblemas tal como estaban enlazados los subproblemas del caso original.

Este proceso se repete recursivamente cuantas veces sea necesario.

Para aquellos subproblemas que no se puedan descomponer, caben dos posibilidades: o bien el sistema realiza una serie de suposiciones pidiendo confirmación al usuario, o pide al usuario que introduzca la solución de ese subproblema, siguiendo el sistema con la resolución del resto.

Finalmente, una vez resuelto el problema inicial, el sistema muestra la solución al usuario, y la indexa en su memoria de la misma forma que indexa los casos de entrenamiento.

Vamos a describir estos procesos de forma más detallada, ayudándonos con un ejemplo.

Supongamos que en la biblioteca de casos tenemos resuelto el problema de contar el número de elementos de una lista enlazada simple de longitud no nula, y se solicita la solución al problema de contar el número de elementos de una lista enlazada circular también de longitud no nula. Como caso más similar, el sistema tomaría el de contar el número de elementos de una lista enlazada simple.

En un primer nivel de refinamiento, el caso recuperado está formado por los bloques de la figura 4.9.

Como se ha dicho, para cada uno de estos bloques, el sistema buscaría otro similar, pero cambiando lista enlazada simple por circular en aquellos lugares donde aparezca. Por ejemplo, para:

```
ACCION tipo INICIALIZAR
       objeto CONTADOR
       cantidad UNO
```

el sistema busca un CD igual, ya que en él no influye el hecho de que el contador lleve el cómputo de nodos de una lista simple o circular. Como este caso ya está resuelto en la memoria de casos, lo encontraría. Se trataría del caso trivial.

Por el contrario, para el CD:

```
ACCION tipo APUNTAR
       objeto PUNTERO
       a PRIM-ELEMTO de OBJETO esun COLECCION-ORGANIZADA
                                num-anteriores UNO
```

```
                              num-siguientes UNO
                              implementada ENLAZADA
                              forma SIMPLE
```

el sistema trataría como caso de entrada el siguiente problema:

```
ACCION tipo APUNTAR
       objeto PUNTERO
       a PRIM-ELEMTO de OBJETO esun COLECCION-ORGANIZADA
                              num-anteriores UNO
                              num-siguientes UNO
                              implementada ENLAZADA
                              forma CIRCULAR
```

es decir, en el lugar de lista enlazada simple ha puesto lista enlazada circular.

Como ya se ha indicado, cada uno de los bloques resultantes se enlazan de la misma forma en que estaban en la solución del problema que está siguiendo. En este ejemplo, se enlazarían los dos CDs obtenidos de forma secuencial. Es decir, del *nuevo* CD:

```
ACCION tipo INICIALIZAR
       objeto CONTADOR
       cantidad UNO
```

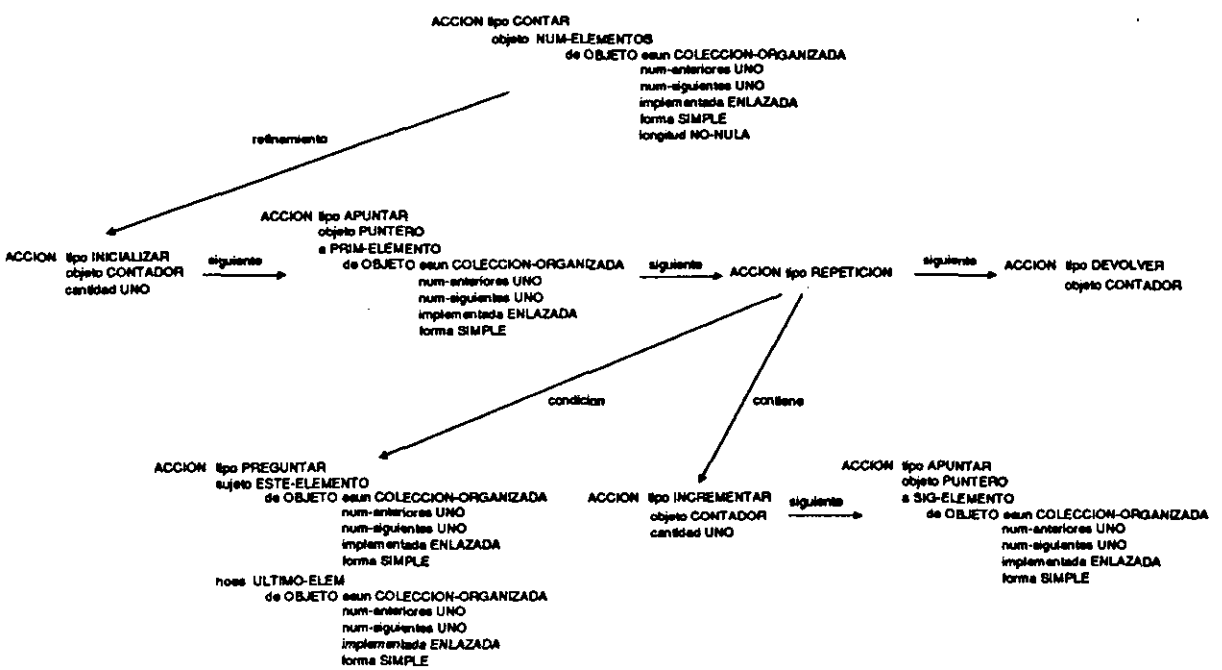

Figura 4.9

saldría un enlace rotulado como "siguiente" al *nuevo* CD:

```
ACCION tipo APUNTAR
       objeto PUNTERO
       a PRIM-ELEMTO de OBJETO esun COLECCION-ORGANIZADA
                                num-anteriores UNO
                                num-siguientes UNO
                                implementada ENLAZADA
                                forma CIRCULAR
```

Hemos recalcado el hecho de que es nuevo, porque el sistema no toma los CDs que encuentra, sino que hace una copia de los mismos. Si simplemente se añadieran más enlaces a los CDs encontrados, llegaríamos a tener CDs con varios enlaces a "siguiente", por ejemplo. En estas situaciones, al seguir la secuencialidad de un programa nos encontraríamos con varias posibilidades y no se sabría qué "siguiente" bloque corresponde a cada uno de los algoritmos de los que forma parte ese CD.

A la hora de buscar un CD en memoria con una serie de características concretas, el sistema trata distinto a los bloques según sean nodos-sentencia secuenciales, nodos-sentencia no secuenciales, o nodos-condiciones:

a) *Sentencias secuenciales:* Intenta buscar un bloque o CD con las características que se desean (tal como se ha explicado anteriormente).

Si lo encuentra, lo toma y lo enlaza -a los subproblemas que ya ha resuelto- con el mismo tipo de enlace que hay en el problema original. Lo toma con todos sus enlaces -excepto el enlace "siguiente"-, es decir, si es un bloque que se puede descomponer en otros más simples (tiene un enlace "refinamiento") lo toma con todos esas componentes más simples hasta sus componentes terminales. Asumimos que si el bloque encontrado reune todas las características buscadas, su descomposición en subproblemas es válida hasta el último nivel de refinamiento.

Si no lo encuentra, intenta buscar analogías en el siguiente nivel (inferior) de refinamiento. Para ello, mira si ese bloque tiene un enlace "refinamiento" (es decir, si esa componente se puede descomponer en operaciones más simples). Si lo tiene, realiza esta misma operación para cada uno de los bloques en que se descompone (es decir trata ese subproblema como si fuera el problema original).

Veamos ejemplos para las dos posibilidades que acabamos de explicar.

Estaríamos en la primera situación si se buscase el CD:

```
ACCION tipo INICIALIZAR
       objeto CONTADOR
       cantidad UNO
```

El sistema lo encontraría, ya que esta acción no depende de la estructura de datos sobre la que se trabaja, y este CD se introdujo en la memoria al indexar el caso que se ha tomado como el más análogo al que se desea resolver. Este CD lo tomaría, en realidad lo *copiaría*, con su enlace "refinamiento" hasta el último nivel.

Supongamos ahora, que lo que se ha pedido es obtener la solución para el enunciado "Contar el número de elementos de una lista enlazada circular de longitud indeterminada". El caso más similar lo tendríamos para lista enlazada simple. Para resolver el problema planteado, seguimos como guía este caso más similar: el de la lista enlazada simple. La solución de este problema se divide en dos, cuando la lista es vacía y cuando no lo es.

Cuando la lista no es vacía, el problema que hay que resolver es "contar el número de elementos de una lista enlazada simple de longitud no nula". Como lo que queremos es resolver este problema para lista enlazada circular, lo que hacemos es buscar en la memoria el CD:

```
ACCION tipo CONTAR
       objeto NUM-ELEMENTOS de OBJETO esun COLECCION-ORGANIZADA
                                      num-anteriores UNO
                                      num-siguientes UNO
                                      implementada ENLAZADA
                                      forma CIRCULAR
                                      longitud NO-NULA
```

Este CD no lo encontraríamos. El sistema mira si existe un enlace "refinamiento" en el CD que nos ha servido de guía (el de contar el número de elementos de una lista enlazada simple de longitud no nula). El enlace existe (ver figura 4.9), así pues, busca de nuevo en la memoria cada uno de los bloques en los que se descompone sustituyendo lista enlazada simple por lista enlazada circular. Cada uno de estos nuevos sub-problemas se tratan como si fueran el problema inicial que propone el usuario.

Existe una tercera posibilidad: la componente no se ha encontrado y no se puede descomponer en otras más sencillas. Entendemos que no se puede descomponer en otras más sencillas cuando o no tiene enlace de refinamiento, o si lo tiene, conduce al último nivel de refinamiento (directamente traducible a código).

Como se ve, en este proceso de ir bajando de nivel de refinamiento para encontrar bloques, se excluye el último nivel de refinamiento. Esto se debe a que en este nivel, ya no se trabaja en términos abstractos, sino que se tratan aspectos concretos de implementación.

A nivel de código no tendría sentido la búsqueda de analogías, ya que para poder razonar sobre los problemas hay que hacerlo a un nivel más abstracto.

En la tercera posibilidad que hemos mencionado, la componente depende del objeto; de lo contrario, valdría la componente misma que está sirviendo de guía. Para obtener esa misma componente pero para otro objeto, el sistema utiliza la información que se le ha suministrado sobre objetos en el diccionario de conceptos.

En el diccionario, y partiendo del objeto para el cual se quiere encontrar una determinada componente, el sistema sube por la jerarquía para bajar por otra rama. De esta manera, obtiene objetos 'hermanos' del inicial. Para estos objetos se busca la componente deseada.

Es decir, por ejemplo, si se busca algo para lista enlazada circular y no se encuentra, como por la jerarquía se puede saber que lista circular es una lista y lista simple también lo es, se busca para una lista simple.

Si aún así, no encuentra esa componente, seguiría subiendo por la clasificación, para bajar de nuevo obteniendo objetos 'primos'. Este proceso pararía cuando o bien se encuentra la componente para un objeto, o no se tiene más información sobre objetos (se ha agotado la clasificación), o bien porque el objeto obtenido en el diccionario difiere más del que se necesita que el que se tiene en el caso que se está utilizando como modelo.

De todo este proceso, se habrá obtenido una componente; sin embargo, como dicha componente no emparejará totalmente con la que se necesita, e incluso puede que haya diferencias importantes, el sistema pedirá confirmación al usuario.

Veámoslo con un ejemplo. Supongamos que en algún punto del proceso, el sistema necesita saber cómo se avanza un puntero cuando tengo una lista enlazada circular:

```
ACCION tipo ASIGNAR
       objeto PUNTERO
       a PRIM-ELEMENTO de OBJETO esun COLECCION-ORGANIZADA
                                 num-anteriores UNO
                                 num-siguientes UNO
                                 implementada ENLAZADA
                                 forma CIRCULAR
```

y que en la memoria no se ha encontrado. El siguiente refinamiento es ya el último nivel; así pues, el sistema utiliza el diccionario de conceptos para buscar el mismo subproblema para un objeto hermano (el más cercano posible), en este caso la lista enlazada simple. El sistema busca y encuentra el siguiente CD:

```
ACCION tipo ASIGNAR
       objeto PUNTERO
       a PRIM-ELEMENTO de OBJETO esun COLECCION-ORGANIZADA
                                num-anteriores UNO
                                num-siguientes UNO
                                implementada ENLAZADA
                                forma SIMPLE
```

Como posible solución al subproblema que está intentando resolver (avanzar el puntero en una lista enlazada circular) tomará el refinamiento de este CD que ha encontrado. Antes de darlo por válido pide la confirmación del usuario.

b) *Sentencias no secuenciales:* Ya se dijo que en los nodos en los que no tenemos un flujo secuencial, el nodo padre del sub-árbol, lo único que indica es el tipo de flujo, y que de éste salen unos enlaces a la condición y sentencias a ejecutar en su caso.

Lo que se hace en estos casos es respetar el hecho de que ahí hay un bucle o una selección. Para ello, el sistema copia el CD que indica el tipo de flujo. A continuación, las condiciones y sentencias que están enlazadas a este CD-padre, se tratan como condiciones o sentencias respectivamente. Una vez resueltas, se ligan al CD-padre copiado con los mismos enlaces del CD original.

c) *Condiciones:* Se tratan igual que los bloques secuenciales en cuanto se refiere a la búsqueda del más similar, a la forma de tratar los bloques que se refinan, y a la forma de tomar todo el refinamiento de un bloque dado. Difieren en que las condiciones se relacionan entre ellas con enlaces de tipo lógico ("y-logico", "o-logico", etc.). Por tanto, en vez de seguir los enlaces "siguiente", se siguen los enlaces "y-logico", "o-logico", etc.

Analogía Aproximada

Nos encontramos en estas situaciones cuando no se ha encontrado ningún caso con analogía directa en la operación. En estas circunstancias, para resolver un caso lo que haremos será recuperar un conjunto de casos -no sólo uno- con analogía aproximada en la operación. La forma de obtenerlos se describe en el siguiente apartado.

A partir de este conjunto de casos, se intenta obtener una solución por dos mecanismos distintos. Primero se procura conseguir un patrón común -con variables pc-var- a todos los casos recuperados. El patrón comprende el enunciado del problema y el primer nivel de refinamiento. Si se consigue, se particulariza para el caso concreto en el que estemos.

Si no se consigue, se intenta generar un esquema general para ese grupo de programas; este esquema está formado por los bloques comunes a todos los casos recuperados, y servirá como base para resolver el problema.

Estos mecanismos se pueden aplicar tanto en el problema inicial que presenta el usuario como en los subproblemas en los que el sistema divide el problema original. Si ninguno de los dos mecanismos anteriores se pudiera aplicar a alguno de los subproblemas, el sistema pide al usuario que proporcione alguna solución a dicho problema, para poder seguir con la resolución de los otros subproblemas.

A continuación se exponen con más detalle estos procesos y la forma de obtener el conjunto de casos con analogía aproximada en la operación.

Obtención de un conjunto de casos análogos

Lo primero que se hace es determinar de qué clase es la operación del caso de entrada, ya que lo que se exige para la analogía aproximada es que las operaciones sean de la misma clase. Esto se obtiene en el diccionario de conceptos.

A continuación, se busca el lugar de la memoria donde se debería situar el CD que representa el enunciado del caso a resolver, teniendo en cuenta la clase a la que pertenece. Como conjunto de casos inicial, tomo aquellos que serían sus 'hermanos'; es decir, aquéllos que son una especificación de la misma abstración de la que se colgaría el caso que estamos resolviendo.

Se procurará que esos casos tengan una analogía directa en el objeto. El número de casos que el sistema obtenga con estas características debe ser mayor que uno. El motivo es que no se dispone de un caso lo suficientemente análogo como para construir la solución sólo a partir de él, por lo que se requieren al menos dos casos. Si sólo encuentra uno o ninguno, el sistema tomará todos los casos 'hermanos' que tenga en memoria con analogía aproximada en la operación y en el objeto.

Si aún así no alcanza la cifra de dos casos, subiría por la jerarquía de abstración al siguiente nivel, para tomar todos los 'primos'del caso que estamos tratando. Este proceso de subir por la abstracción para ampliar el número de casos se repetiría hasta conseguir un número de casos análogos superior a dos. En esta ascensión por la jerarquía hay que tener en cuenta que los casos que se cojan deben tener analogía aproximada en la operación. Si no se encontrasen casos con estas características, el sistema avisaría de que no encuentra casos análogos y no puede resolver el problema. En principio hay que suponer que la biblioteca de casos estará suficientemente llena y no se llegará a estas situaciones.

Finalmente, queremos hacer notar que, tal como se seleccionan los casos, siempre existirá una abstracción común a todos ellos que denominaremos abstracción 'madre' u 'origen'.

Proceso de resolución por creación de patrones

En el capítulo precedente, al hablar de la indexación de información compleja, se introdujo el concepto de pc-var: variables que sustituyen aquellos CDs que aparecen repetidamente como el valor (*filler*) del mismo *slot* dentro de un CD complejo.

Esta misma noción se puede aplicar a redes de CDs, y por tanto también al árbol de solución de un caso. Debido a la forma de nuestro árbol de refinamientos -en el que un nodo es el enunciado de un problema, y sus hijos los subproblemas en los que se descompone el anterior- las variables pc-var, se buscarán sólo entre un nodo y su primer nivel de refinamiento.

Estas pc-var se pueden buscar tanto para los *fillers* del *slot* "tipo" (que señala la operación), como para los *fillers* del *slot* "objeto".

Normalmente, las pc-var para los valores del *slot* "objeto" se encontrarán fácilmente, ya que si en el enunciado se habla de un determinado objeto, éste aparecerá en el primer nivel de refinamiento. Para nosotros tiene más interés localizar pc-var en el resto de los *slots*.

Dado un CD enunciado y su primer nivel de refinamiento, llamaremos *patrón* a la red de CDs que se obtiene al sustituir los *fillers* de los *slots* por las variables pc-var correspondientes, si éstas se pueden establecer.

Encontrar un grupo de problemas para los cuales el patrón que se obtiene es el mismo -idéntico-, indica que existe un método de resolución común a ese grupo de problemas.

Volviendo con el método de resolución, nos encontramos en una situación en la que tenemos un grupo de casos con analogía aproximada en la operación, y directa o aproximada en el objeto -según las circunstancias-, y además, estos casos son todos 'hijos', o 'nietos', etc. de una misma abstracción.

Para cada uno de estos casos recuperados de la memoria se obtiene su patrón. Si el patrón obtenido para cada uno de los casos es el mismo, se supone que también es aplicable al caso concreto que tenemos a resolver, ya que -al igual que el resto de los casos a partir de los cuales se generó el patrón- es una particularización de una abstracción común.

El patrón se aplica a un caso, ligando todas las variables pc-var del patrón según lo marcan los *fillers* o valores del CD enunciado que se está resolviendo.

Con este proceso, se ha pasado al primer nivel de refinamiento, es decir, tenemos ahora una serie de subproblemas a resolver. Cada uno de estos subproblemas se resuelven tomándolos como si fueran casos de entrada.

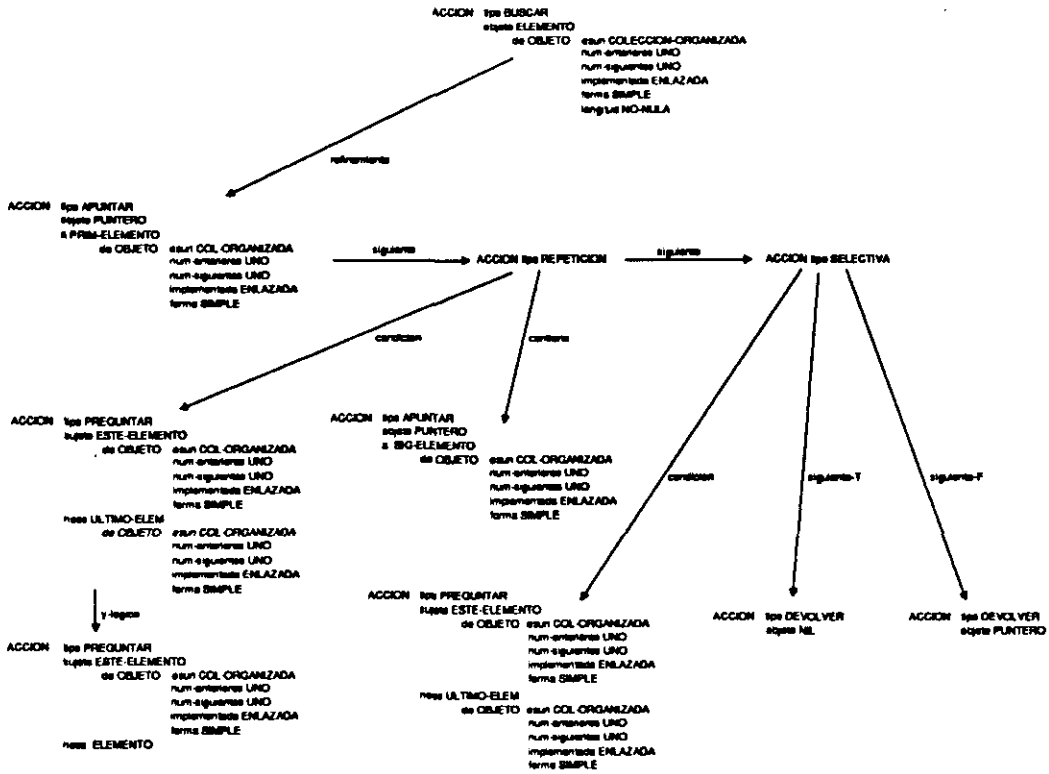

Figura 4.10

El patrón obtenido se enlaza a la abstración 'madre' de todos los casos que sirvieron para obtenerlo con un enlace LINK-R rotulado como "patron", y se indexa en la memoria con la función 'indexa-caso', es decir, como si fuera un caso de entrenamiento, ya que realmente un patrón se puede ver como un enunciado y su primer nivel de refinamiento.

Naturalmente en estas situaciones, antes de crear el patrón, se averigua si la abstracción ya tiene construido ese patrón, en cuyo caso lo único que resta es aplicarlo al caso concreto.

Se podría haber optado por crear los patrones en el momento en que se crea la abstracción, al indexar los casos y conceptos en la memoria. Sin embargo, las abstracciones se forman a partir de un número muy pequeño de casos, y por tanto la conclusión sería poco fiable. Por otro lado, puede haber abstracciones que no correspondan a abstracciones de enunciados reales, y por tanto, nunca se usen como tales, por lo que se habría realizado un trabajo inútil.

Vamos a seguir un ejemplo. Supongamos que se pide "Mostrar el contenido de cada uno de los elementos de una lista enlazada simple de longitud indeterminada (M LES Ind) y el sistema dispone de los programas de Contar (ver figura 4.9) y Buscar (se describe en la figura 4.10) en una lista enlazada simple de longitud indeterminada (C LES Ind, B LES Ind).

El CD que representa el problema de mostrar es:

```
ACCION tipo MOSTRAR
        objeto CONT-CADA-ELEMENTO de OBJETO esun COLECCION-ORGANIZADA
                                                num-anteriores UNO
                                                num-siguientes UNO
                                                implementada ENLAZADA
                                                forma SIMPLE
                                                longitud INDETERMINADA
```

A partir de C LES Ind y B LES Ind obtengo el siguiente patrón: En los dos casos se trata de determinar lo primero la longitud de la lista, dividiendo el problema en dos: cuando la lista es vacía y cuando no lo es. Realmente, el patrón obtenido por el sistema se corresponde con el método habitual que emplean los programadores para solucionar la mayoría de los problemas de listas.

El CD enunciado del patrón es:

```
ACCION tipo ?TIPO
       objeto ?OBJETO de OBJETO esun COLECCION-ORGANIZADA
                                num-anteriores UNO
                                num-siguientes UNO
                                implementada ENLAZADA
                                forma SIMPLE
                                longitud  INDETERMINADA
```

Su resolución consiste en preguntar si la lista está vacía o no para dividir el problema en dos. Este CD condición no tiene variables pc-var:

```
ACCION tipo PREGUNTAR
       objeto ESTADO
              sujeto LONGITUD de OBJETO esun COLECCION-ORGANIZADA
                                            num-anteriores UNO
                                            num-siguientes UNO
                                            implementada ENLAZADA
                                            forma SIMPLE
              es NULA
```

si es vacía se ejecuta:

```
ACCION tipo ?TIPO
       objeto ?OBJETO de OBJETO esun COLECCION-ORGANIZADA
                                num-anteriores UNO
                                num-siguientes UNO
                                implementada ENLAZADA
```

```
                                    forma SIMPLE
                                    longitud NULA
```

si no es vacía, se ejecuta:

```
ACCION tipo ?TIPO
       objeto ?OBJETO de OBJETO esun COLECCION-ORGANIZADA
                                num-anteriores UNO
                                num-siguientes UNO
                                implementada ENLAZADA
                                forma SIMPLE
                                longitud NO-NULA
```

Las variables del patrón se ligan según el caso a resolver. Así, ?TIPO se liga a MOSTRAR, y ?OBJETO a CONT-CADA-ELEMENTO. El problema se ha quedado reducido a dos más pequeños: M LES Vac y M LES No Vac, que se tratarán como si fueran casos de entrada.

```
ACCION tipo MOSTRAR
        objeto CONT-CADA-ELEMENTO de OBJETO esun COL-ORGANIZADA
                                             num-anteriores UNO
                                             num-siguientes UNO
                                             implementada ENLAZADA
                                             forma SIMPLE
                                             longitud NULA

ACCION tipo MOSTRAR
        objeto CONT-ELEMENTOS de OBJETO esun COL-ORGANIZADA
                                           num-anteriores UNO
                                           num-siguientes UNO
                                           implementada ENLAZADA
                                           forma SIMPLE
                                           longitud NO-NULA
```

Finalmente, este patrón se enlaza a la abstracción 'madre' de Contar y Buscar -Recorrer-, y se indexa en memoria, quedando tal como muestra la figura 4.11. De nuevo, ni se han tenido en cuenta todos los índices, ni se muestran todas las abstracciones intermedias.

Proceso de resolución por creación de esquemas

Un método de resolución de problemas bastante utilizado consiste en la identificación, dentro de cada nuevo problema, de un conjunto de características comunes con otros problemas semejantes ya resueltos. Posteriormente, se intenta aplicar al nuevo problema el método abstracto utilizado en problemas previos, añadiendo, suprimiendo o modificando

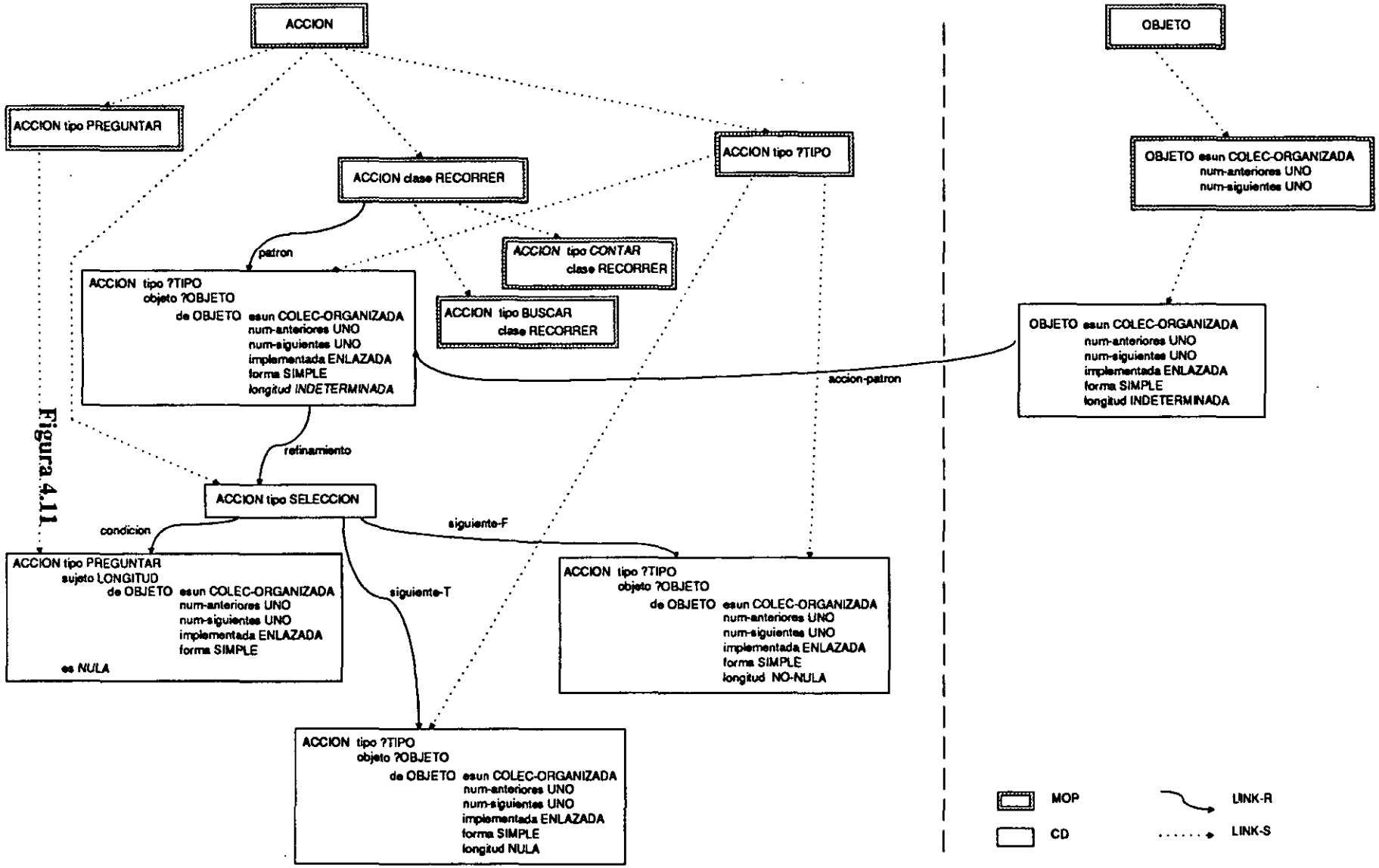
ACCION
ACCION tipo PREGUNTAR
ACCION clase RECORRER
ACCION tipo ?TIPO
patron
ACCION tipo CONTAR
clase RECORRER
ACCION tipo BUSCAR
clase RECORRER
ACCION tipo ?TIPO
objeto ?OBJETO
de OBJETO esun COLEC-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
implementada ENLAZADA
forma SIMPLE
longitud INDETERMINADA
refinamiento
ACCION tipo SELECCION
condicion
siguiente-F
siguiente-T
ACCION tipo PREGUNTAR
sujeto LONGITUD
de OBJETO esun COLEC-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
implementada ENLAZADA
forma SIMPLE
es NULA
ACCION tipo ?TIPO
objeto ?OBJETO
de OBJETO esun COLEC-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
implementada ENLAZADA
forma SIMPLE
longitud NO-NULA
ACCION tipo ?TIPO
objeto ?OBJETO
de OBJETO esun COLEC-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
implementada ENLAZADA
forma SIMPLE
longitud NULA
OBJETO
OBJETO esun COLEC-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
OBJETO esun COLEC-ORGANIZADA
num-anteriores UNO
num-siguientes UNO
implementada ENLAZADA
forma SIMPLE
longitud INDETERMINADA
accion-patron
MOP
CD
LINK-R
LINK-S

Figura 4.11

algunas partes de dicho método. Esta es la idea de fondo en este proceso de resolución por creación de esquemas.

Esquema es la representación de algo atendiendo sólo a sus líneas o caracteres más significativos. Un esquema de programa puede considerarse como representante de una familia o clase de programas reales (Fernández Chamizo, 1984).

Aunque normalmente a la noción de esquema -junto con trozos comunes- va unida la existencia de puntos de inespecificación que se concretan, para cada programa en particular, de acuerdo con unas ligaduras (Rich, 1981), en nuestro prototipo, por esquema de programa nos referimos a aquellas sentencias -secuenciales o no, bloques o sentencias propiamente dichas- comunes a todos los programas que forman parte de una misma clase. Consiste por tanto en una abstracción de características y estructura comunes a varios programas.

No han de confundirse estas abstracciones de características comunes entre dos programas, con las abstracciones que genera el sistema: los MOPs. Los esquemas son una serie de bloques a ejecutar para resolver un problema. Los MOPs son abstracciones entre bloques individuales, es decir, abstracciones en las características de los enunciados de los problemas, no en su algoritmo de resolución.

Estos bloques o componentes comunes a un conjunto de operaciones sólo se buscan en un mismo nivel de refinamiento, es decir, en el primer nivel de refinamiento respecto al nodo-enunciado que se pretende resolver. Si el problema a resolver es el principal, el esquema se busca en el primer nivel del árbol de refinamientos; si se trata de un problema que está en un nivel más bajo en el árbol, el esquema se busca entre los nodos del nivel inmediatamente inferior.

El proceso para generar estos esquemas es el siguiente:

El sistema toma como esquema inicial uno cualquiera de los casos recuperados de la memoria. Este primer esquema, de esquema tiene poco, ya que es un programa concreto con todos sus detalles, pero a lo largo del proceso se le van quitando esos detalles para quedarse sólo con esas características comunes a todos los programas recuperados, hijos de una misma abstracción.

A continuación, y para todo el resto de casos recuperados, toma un caso y lo va comparando con el esquema que tiene en ese momento. Si algún bloque del esquema no se "encuentra" en el caso con el que estamos comparando, se quita del esquema. Para los bloques selectivos o iterativos, se tratan por un lado las sentencias y por otro las condiciones. Tomamos como esquema resultante el esquema que queda una vez se ha comparado con todos los casos recuperados de la memoria.

Por "encuentra" entendemos que el bloque debe estar, pero no en cualquier posición sino en una equivalente; equivalente no quiere decir que tenga que ser exactamente la misma, sino que puede estar antes o después de otro bloque si no son dependientes entre sí, es decir si la salida de uno no influye en la entrada del siguiente.

La dependencia o independencia entre dos bloques se determina a través de los *slots* y *fillers* de los CDs que los representan, que son los que fijan su semántica. En términos generales se puede decir que dos bloques son independientes si las acciones que ejecutan -sean o no las mismas- son sobre objetos distintos. Si dos bloques actúan sobre los mismos objetos, su dependencia o independencia se puede determinar por los *slots* "precondición" y "poscondición" que existen en los bloques del último nivel de refinamiento.

Cuando existen bloques selectivos o iterativos, como resultado de las comparaciones nos podemos encontrar en las siguientes situaciones:

1. Queda al menos un bloque de condición y uno de sentencia. Se mantiene el hecho de que ahí hay una selección o repetición con las condiciones y sentencias que han quedado.

2. Sólo han quedado bloques condición y ninguno de sentencia. En el lugar de ese bucle o selección no se pone nada.

3. Sólo han quedado componentes sentencias. Se quita el hecho de que hay una repetición o selección y se dejan en su lugar las sentencias que han quedado.

Si el resultado obtenido no consta de ningún bloque, entonces no tenemos esquema y no podemos aplicar este método. El sistema lo notifica al usuario y pide que introduzca la solución.

De forma similar al proceso con patrones, el esquema resultante se enlaza con la abstracción 'madre' con un enlace LINK-R rotulado como "esquema", y se indexa en la memoria. Antes de realizar todo este proceso, se miraría que la abstracción no tuviera ya el esquema generado.

Siguiendo con el ejemplo que proponíamos en el apartado anterior, supongamos que ahora tenemos que resolver el problema de "Mostrar el contenido de cada nodo de una lista enlazada no vacía" (M LES No Vac), y que en la biblioteca de casos tenemos "Contar el número de nodos de una lista enlazada no vacía" (C LES No Vac) y "Buscar un elemento determinado en una lista enlazada simple no vacía" (B LES No Vac).

Para los dos casos recuperados C LES No Vac y B LES No Vac, no existe un patrón que se pueda ejemplificar a nuestro caso concreto. Intentamos, por tanto, obtener un esquema.

Como esquema inicial se toma uno cualquiera, por ejemplo la solución de C LES No Vac, ya descrito en la figura 4.9.

A continuación se toma otro de los casos recuperados: B LES No Vac (la solución puede verse en la figura 4.10), en esta situación concreta, el único que queda.

Componente a componente, se va comparando el esquema que por ahora tengo, la solución para C LES No Vac, con el caso que he tomado.

El esquema tiene un bloque en el que se inicializa un contador. El caso con el que lo comparo no lo tiene, por lo que se quita este bloque del esquema.

La componente por la que un puntero apunta al primer elemento de la lista la tienen los dos. Están también en una posición similar ya que el bloque que inicializaba el contador se ha quitado y además apuntar al principio de la lista e inicializar un contador son operaciones independientes.

El siguiente bloque indica la presencia de una repetición, por lo que compararemos los bloques colgados del enlace "condicion" entre sí, y los colgados del enlace "contiene", también entre sí.

Para la condición, en lo que queda de esquema tenemos la comprobación de si se trata del último elemento de la lista; esta componente también se encuentra en B LES No Vac., y por lo tanto se deja. La condición que hay para B LES No Vac. pregunta también si el elemento en el que estoy es justo el que quiero, pero este bloque ni se trata. Lo único que se hace es buscar cada bloque del esquema en el caso, pero no viceversa. Hay que hacer notar que el esquema final resultante es independiente del caso elegido como esquema inicial.

De forma similar, las componentes del esquema de incrementar el contador y de devolver el mismo, no se encuentran en el caso de B LES No Vac., por lo que se quitarían del esquema. El resultado final sería el que se muestra en la figura 4.12.

Siguiendo con el proceso, una vez tengo un esquema, se adjudica como una primera solución al caso a resolver.

Este esquema puede estar descrito para un objeto con analogía directa con el de la entrada, o bien para un objeto con el que se tiene analogía aproximada.

Si se trata de una analogía directa, se toma cada uno de los bloques que lo forman tal cual, con todos sus refinamientos. Si la analogía es aproximada, cada uno de los bloques se trata como si fuera un problema de entrada en el que hay que resolver una analogía directa en la operación, pero aproximada en el objeto.

En este punto, lo que tenemos es un esquema que indica las características comunes a una serie de programas relacionados entre sí. A continuación, lo que se pretende es añadir las partes específicas del problema que estamos intentando resolver. Para esto necesitamos dos datos: en qué operaciones concretas consisten esas partes específicas, y dónde deben situarse dentro del esquema que se acaba de obtener.

A través de las definiciones y reglas que se dieron en el diccionario de conceptos, se obtienen las operaciones específicas del problema que estamos tratando, y que hay que añadir al esquema obtenido. La posición, dentro del esquema, donde deben situarse se obtiene por analogía con los casos que sirvieron para generar el esquema.

El proceso para realizar esto sería el siguiente:

De la definición y reglas de la operación, obtenemos aquellas operaciones en las que -a grandes rasgos- puede descomponerse. Esta información viene dado los los *slots* del CD. Por ejemplo, para CONTAR, tenemos una inicialización -dada por el *slot* "ini-tipo", una operación a realizar sobre cada elemento de la estructura: incrementar el contador -dada por el *slot* "cada-elem-tipo", y un resultado, el contador -dado por el *slot* "resultado"-. Para el caso de MOSTRAR, sólo tendríamos una suboperación a realizar sobre cada elemento de la estructura: mostrar el contenido del nodo -también dado por el *slot* "cada-elem-tipo".

A continuación, para cada una de estas suboperaciones, se localiza por analogía, la posición en el esquema. Estas suboperaciones vienen detarminadas, cada una, por un *slot* que indica el papel que juegan dentro de la operación global (inicializar, devolver resulta-

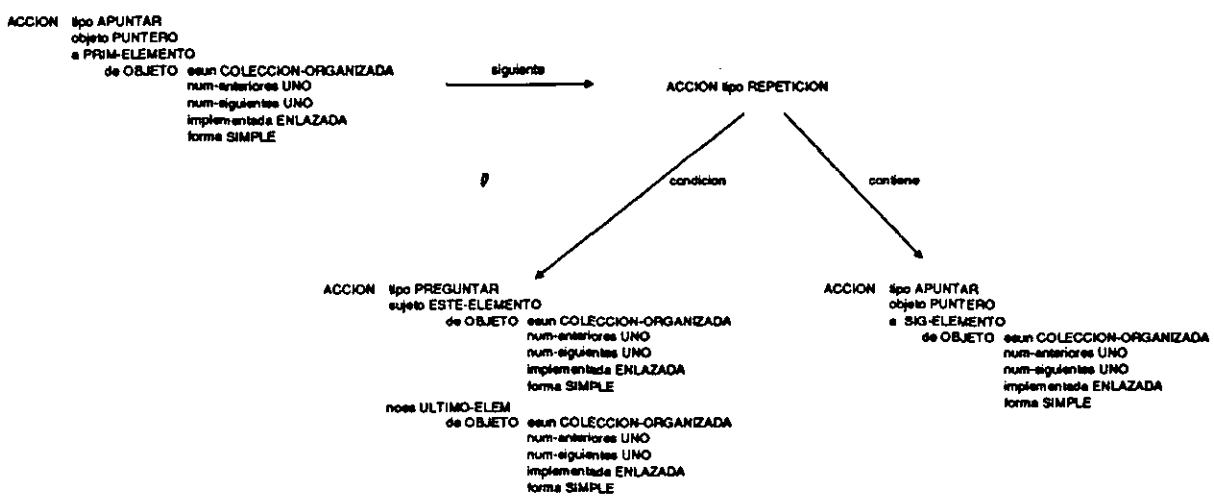

Figura 4.12

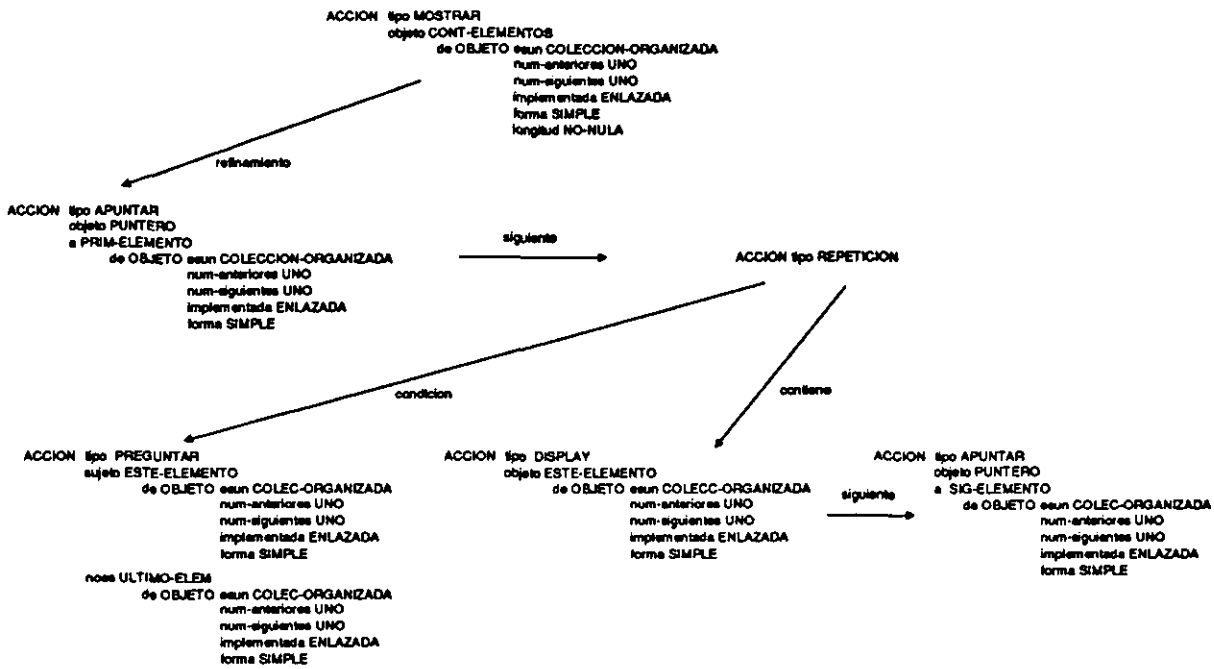

Figura 4.13

do,etc). De entre los casos que sirvieron para generar el sistema, se coge el que mejor empareje con el caso a resolver por los métodos descritos anteriormente. Se obtiene qué tipo de operación resuelve ese caso, y se mira si en su regla de descomposicón también existe una suboperación con la misma funcionalidad (inicializar, etc) que la que estoy tratando. En caso afirmativo, se determina su posición en el árbol de descomposición de la solución, y ésta se asigna a la suboperación, pidiendo previamente confirmación al usuario, que tiene la oprtunidad de modificarla. Si esta suboperación no se encontrase en el caso que mejor empareja, se tomaría el siguiente caso que mejor empareje. Este proceso seguiría hasta que se encontrase un caso con esa suboperación. De no encontrarse ninguno, se pediría al usuario que indique la posición de la misma.

Hay que tener en cuenta que las suboperaciones que se describen en las reglas, vienen dadas en un nivel alto de abstracción. Para obtener sus sucesivos refinamientos, hay que buscar esta componente en la memoria de casos. Si se encuentra, se toman todos sus niveles de refinamiento hasta el nivel directamente traducible a código. En caso contrario, se pide al usuario que especifique esos niveles.

Finalmente, se comprueba que las precondiciones y poscondiciones de los bloques se satisfacen. En caso contrario, por medio de reglas se realizan las oprtunas modificaciones. Este último proceso no se ha implementado por el momento.

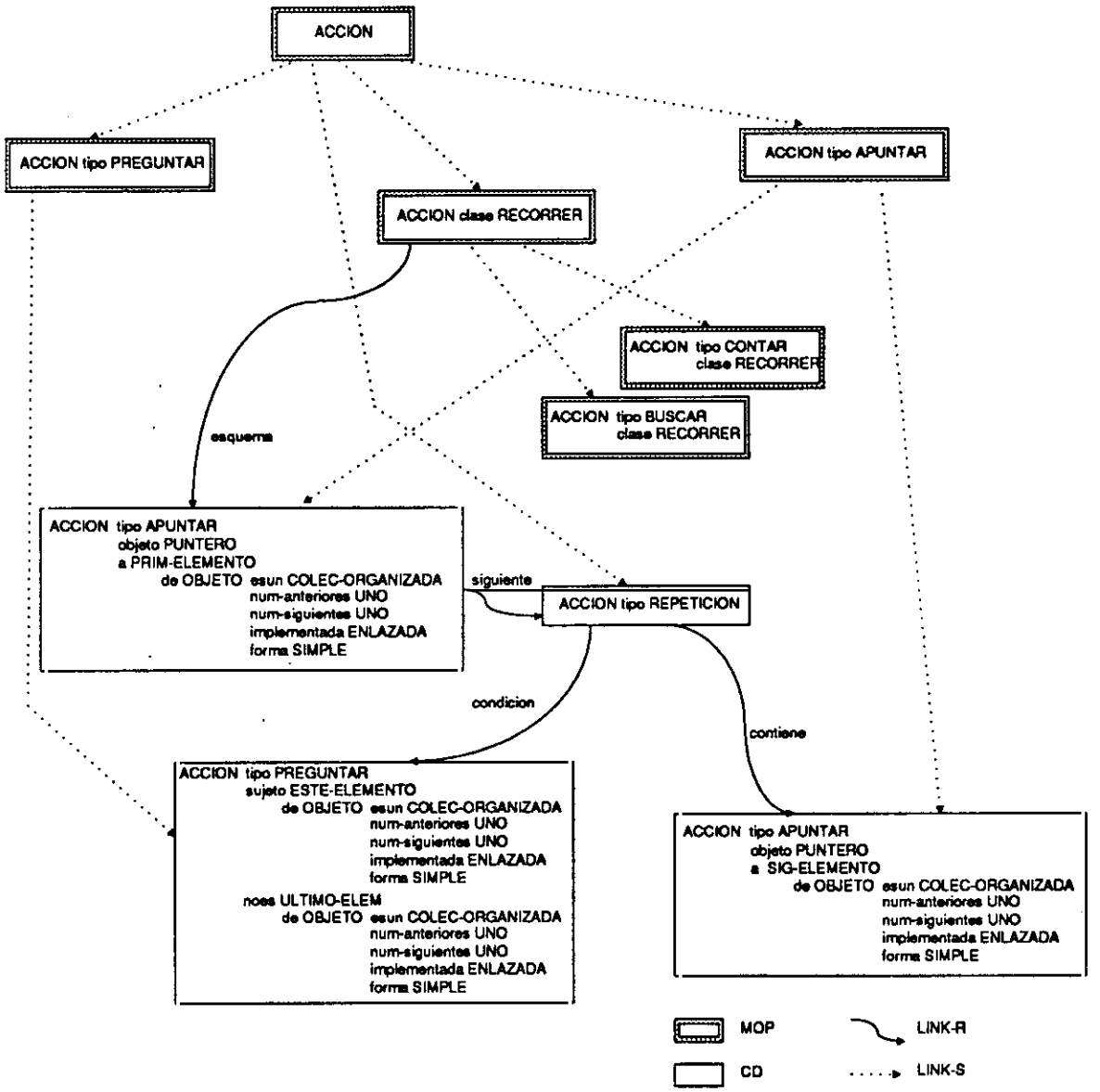

Figura 4.14

Siguiendo con el ejemplo anterior, habíamos encontrado el esquema siguiente: primero se inicializa un puntero apuntando al principio de una lista, luego se tiene una iteración, donde la condición de fin es que se trate del último elemento, y lo que se hace dentro del bucle es apuntar al siguiente elemento.

A continuación, por la definición de la operación MOSTAR, se observa que consiste en una única suboperación que se ejecuta para cada elemento.Esta suboperación viene determinada por el *slot* "cada-elem-tipo". Para el caso de CONTAR también tenemos una suboperación que se ejecuta para cada elemento de la estructura -la determinada por el *slot* "cada-elem-tipo"-. Esta está situada justo antes de la componente de avanzar el puntero;

en esta misma posición se pone la suboperación de mostrar el contenido de un nodo. Como en la memoria no se dispone del refinamiento de esta operación, se pediría el usuario que introdujera esa información.

El problema M LES No Vac, ha quedado resuelto tal como muestra la figura 4.13.

También este esquema se indexaría en la memoria una vez se hubiera enlazado a la abstracción Recorrer. Puede verse en la figura 4.1.

Con este método de resolución por creación de esquemas, hemos terminado de describir el sistema de síntesis de programas que proponemos. En el capítulo próximo veremos las posibles continuaciones.

5. CONCLUSIONES, APLICACIONES Y FUTUROS DESARROLLOS

5.1. CONCLUSIONES Y APLICACIONES

Se ha desarrollado un sistema que permite sintetizar programas por medio de la detección de analogías con otros programas previamente desarrollados. El sistema utiliza un mecanismo de razonamiento basado en casos (CBR) que le permite generalizar a partir de ejemplos en el dominio de la aplicación. El prototipo desarrollado aprende con la experiencia, utilizando el conocimiento adquirido durante el diseño de un algoritmo como guía para el diseño de otros.

El prototipo que presentamos aquí, se desarrolló inicialmente para modelar el proceso de aprendizaje en un dominio de aplicación diferente -la mecánica fundamental- (Fernández-Valmayor & Fernández Chamizo, 1992). Se desarrolló en Allegro Common Lisp. Hemos intentado demostrar la versatilidad del formalismo de representación del conocimiento, aplicando el mismo método a un dominio completamente diferente. En este trabajo hemos mostrado cómo los sistemas CBR pueden ser explotados para permitir el desarrollo de programas por inspección.

En cuanto a las posibles aplicaciones del sistema, tendríamos las siguientes:

- Componente de un sistema de Programación Automática. Como una componente dentro de un sistema global de Programación Automática. Por el momento, el sistema no pretende ser un sistema de programación automática. Los métodos de inspección son el enfoque más efectivo para el diseño de software donde quiera que se aplique. Sin embargo, como los métodos de inspección funcionan sólo si existen problemas análogos en la base de conocimientos, deben ser completados con métodos más generales como los mecanismos de deducción y transformación. Pensamos que experimentar con este prototipo permitirá la especificación de un módulo basado en analogía que formaría parte de un sistema más general de programación automática.

- Simulación de aprendizaje. Por otra parte, el sistema podría ser usado para simular el aprendizaje humano de algunas de las habilidades involucradas en la programación. Los resultados de esta simulación podrían servir como base de nuevos enfoques en la enseñanza de la programación.

- Reutilización de software. Una de las funciones que realiza este sistema es catalogar e indexar problemas con sus soluciones. Tal como se reseñó en el capítulo primero, éstos son dos de los problemas con los que nos encontramos en la reutilización de software. El sistema que proponemos aquí, también es capaz de indexar y catalogar, con un mismo formalismo, objetos. Así pues, los métodos propuestos, se podrían aplicar también para desarrollar herramientas de ayuda a la reutilización de software.

Otro de los problemas con que se encuentra la reutilización de software, como ya vimos, es la elección del tamaño de la componente a reutilizar. Con el método de descripción de problemas y sus soluciones -que hemos propuesto en este trabajo- por niveles de refinamiento-, se puede tanto acceder a un problema global, como a cualquiera de sus subproblemas, ya que están perfectamente especificados y clasificados en la memoria de casos.

5.2 FUTUROS DESARROLLOS

El prototipo ha de ser completado con los módulos pre- y posprocesador. El preprocesador transformará la entrada, expresada en un subconjunto restringido de lenguaje natural, en las estructuras CD requeridas por el prototipo. El preprocesador utilizará la propia la base de conocimientos para obtener el conocimiento del dominio necesario para llevar a cabo esta transformación. Se implementará como un *parser* basado en expectativas y la base de conocimientos dinámica será la fuente de información del contexto. El posprocesador eliminará las partes redundantes obtenidas en la solución y traducirá la solución a un lenguaje de programación imperativo, incorporando algunas partes complementarias (como declaraciones de variables, etc.), juntando bucles y realizando cualquier otra operación de "pulido" de la solución.

Algunos trabajos subrayan las diferencias entre las actividades progresivas (trabajar hacia el objetivo establecido por el problema) y evaluativas (evaluar la parte realizada de la solución del problema) en la resolución de un problema. Nuestro prototipo, como la mayoría de los sistemas desarrollados, ha ignorado el proceso de evaluación, siendo el usuario del sistema el responsable de la evaluación de los programas. La ejecución simbólica a los distintos niveles de abstracción, se tendrá en cuenta en futuros desarrollos.

En sucesivas versiones del prototipo intentaremos suministrar mecanismos de explicación y reparación. El prototipo también deberá ser completado con otros métodos de síntesis, como reglas de producción, que puedan ser utilizadas cuando no existan casos análogos.

Próximamente, vamos a intentar utilizar el prototipo como el núcleo de un tutor de programación inteligente. Compararemos las capacidades de este tipo de tutores frente a las capacidades que presentan los tutores basados en reglas como PROUST (Johnson & Soloway, 1985) y CAPRA (Garijo *et al.*, 1987).

Otro proyecto previsto está relacionado con la aplicación de los métodos de indexación que hemos desarrollado a un sistema de programación orientado a objetos. Este proyecto se centrará en el problema de localización de objetos y métodos en una biblioteca de software, para facilitar la reutilización.

BIBLIOGRAFIA

Abbott, R.J., 1983, "Program Design by Informal English descriptions", Communications of the ACM, Vol. 26, No. 11, November 1983, pp. 882- 894.

Abbott, R.J., 1987, "Knowledge Abstraction", Communications of the ACM, Vol. 30, No. 8, August 1987, pp. 664-671.

Alterman, R., 1986, "An adaptive planner", en *Proceedings of AAAI- 86*, pp. 65-69, American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc., Los Altos, CA.

Anderson, J.A., 1983, "The Arquitecture of Cognition", Cambridge, MA: Harvard University Press.

Arango, G., Baxter, I. & Freeman, P., 1988, "A Framework for Incremental Progress in the Application of Artificial Intelligence to Software Engineering", ACM SIGSOFT, Software Engineering Notes, Vol. 13, No. 1, January 1988, pp 46-50.

Ashley, K.D. & Rissland, E.L., 1988, "A Case-Based Approach to Modeling Legal Expertise", IEEE Expert, Vol. 3, N. 3, Fall 1988, pp. 70-77.

Balzer, R.M., 1972, "Automatic Programming", Technical Report RR-73-1, USC/Information Sciences Institute, Marina del Rey, California.

Barlett, F.C., 1932, "Remembering. A Study in Experimental and Social Psychology", Cambridge University Press (2a. reimpresión, 1977).

Barnes, J.G.P., 1982, "Programming in Ada", Addison-Wesley Publishing Company. London.

Barnett, J., Knight, K., Mani, I & Rich, E., 1990, "Knowledge and Natural Language Processing", Communications of the ACM, Vol. 33, No. 8, August 1990.

Barr, A., Feigenbaum, E.A. & Cohen P.R., 1981, "The Handbook of Artificial Intelligence", 3 vol., William Kaufmann, Inc., Los Altos, CA.

Barstow, D.R., 1979, "An Experiment in Knowledge-based Automatic Programming", Artificial Intelligence, Vol. 12, pp. 73-119.

Barstow, D.R., 1979, "Knowledge-Based Program Construction", Thomas E. Cheatham, ed., Elsevier North-Holland, New York.

Barstow, D.R., 1984, "A Perspective on Automatic Programming". AI Magazine, Spring 1984, pp. 5-27..

Bassett, P.G., 1987, "Frame-Based Software Engineering"; IEEE Software, Vol. 4, July 1987, pp. 9-16.

Bhansali, S., 1991, "Domain-Based Program Synthesis Using Planning and Derivational Analogy", Ph.D., Department of Computer Science. University of Illinois at Urbana-Champaing, 1991.

Biermann, A.W., 1972, "On the Inference of Turing Machines from Sample Computations", Artificial Intelligence, Vol. 3, pp. 181-198.

Biermann, A., 1987, "Automatic Programming", Encyclopedia of AI, 1987 Shapiro, S.C. editor in Chief.

Biggerstaff, T. & Perlis, A. J., 1984, "Foreword" al *Special issue on Software Reusability*, IEEE Transactions on Software Engineering,. Vol. SE-10, No. 5, September 1984, pp. 474-477.

Böhm, C. & Jacopini, G., 1966, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", Communications of the ACM, Vol. 9, No. 5, May 1966, pp. 366-371.

Booch, G., 1983, "Software Engineering with Ada", ed. Benjamin- Cummings, 1983.

Broy, M, 1983, "Program Construction by Transformations: A Family Tree of Sorting Programs", en Biermann, A. W. & Guiho, g. (eds), *Computer Program Synthesis Methodologies*,pp. 1-49, D. Reidel Publishing Company.

Buchanan, B.G., 1983, "Partial Bibliography of Work on Expert Systems", ACM-SIGART Newsletter, Vol. 84, pp. 45-50.

Buchanan, B.G. & Feigenbaum, E., 1978, "DENTRAL and Meta-DENTRAL:their Application Dimension", Artificial Intelligence, Vol. 11, pp. 5-24.

Buzzard, G.D., & Mudge, T.N., 1985, "Object-Based Computing and the Ada Programming Language", Computer, Vol. 18, No. 3, March 1985, pp. 11-19.

Caldiera, G. & Basili, V., 1991, "Identifying and Qualifying Reusable Software Components", Computer, Vol. 24, No. 2, February 1991, pp 61- 70.

Carbonell, J.G., 1986, "Derivational Analogy: A Theory of reconstructive problem solving and expertise acquisition, en Michalsh, R.S., Carbonell, J.G. & Mitchell, T.M. (eds.), Machine Learning. An Artificial Intelligence Approach, 2, Morgan Kaufmann, Los Altos, CA.

Charniak, E, Riesbeck, C.K., McDermott, D. & Meehan, J.R., 1987, "Artificial Intelligence Programming Techniques", Lawrence Erlbaum Asociates, Hillsdale, N.J.

Cheng, T.T., Lock, E.D. & Prywes, N.S., 1984, "Use of Very High Level Languages and Program Generation by Management Professionals", IEEE Transacions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 552-563.

Cohen, J., 1985, "Describing Prolog by Its Interpretation and Compilation", Communications of the ACM, Vol. 28, No. 12, December 1985, pp. 564-574.

Cuena, J., Molina, M. & Alonso, M., 1989, "CYRAH: Un Ejercicio de Integración de Modelos de Simulación y Reglas para Representación del Comportamiento Físico", Actas de la III Reunión Técnica de la Asociación Española para la Inteligencia Artificial, Madrid, Noviembre de 1989, pp. 173-192.

DARPA, 1989, "Case-Based Reasoning", Proceedings of Case-Based Reasoning Workshop, May, 1989, Morgan Kaufmann Publishres, Los Altos, CA..

Davies, S.P., 1991, "The Role of Notation and Knowledge Representation in the Determination of Programming Strategy: A Framework for Integrating Models of Programming Behavior", Cognitive Science, Vol. 15, pp. 547-572.

Davies, R., Buchanan, B.G., & Shortliffe, E., 1977, "Production Rules as a Representation for a Knowledge-Based Consultation Program", Artificial Intelligence, Vol. 8, No. 1, ? 1977, pp. 15-45.

DeJong, G. & Mooney, R., 1986, "Explanation-based learning: an alternative view", Machine Learning, Vol. 1, No. 2, 1986, pp. 145- 176.

Dershowitz, N., 1986, "Programming by analogy", en Michalski, R.S., Carbonell, J.G. & Mitchell, T.M., *Machine Learning: An Artificial Intelligence Approach,* Vol. II, ed. Morgan Kauffman, 1986.

Ernst, G.W. & Newell, A., 1969, "GPS: A Case Study in Generality and Problem Solving", Academic Press, New York, 1969.

Fernández Chamizo, C., 1984, "Métodos de Reutilización de Software Basados en Esquemas de Programas". Tesis Doctoral, Madrid, 1984.

Fernández- Valmayor, A., 1990, "Diseño de una base de conocimientos y su aplicación en un entorno educativo", Tesis Doctoral, Universidad Complutense de Madrid.

Fernández-Valmayor, A., & Fernández Chamizo, C.,1992, "Educational and Research Utilization of Dynamic Knowledge Base", Computers & Education, Vol. 18, No. 1-3, ? 1992, pp. 51-61.

Fikes, R.E. & Nilson, N.J., 1971, "STRIPS: A new approach to the applications of theorem proving to problem solving", Artificial Intelligence, Vol. 2, 1971, pp. 189-208.

Freeman, P., 1983, "Reusable Software Engineering: Conceps and Research Directions". ITT Proceedings of the Workshop on Reusability in Programming, pp. 2-16, en Freeman, F., 1987, *Tutorial: Software Reusability*, pp. 10-23, IEEE Computer Society Press.

Frost, R., 1986, "Introduction to Knowledge Base System". Ed. Collins. London.

Gargaro, A. & Pappas, F., 1987, "Reusability Issues and Ada". IEEE Software, Vol. 4, No. 7, July 1987, pp. 43-51.

Garijo, F., Verdejo, F., Díaz-Ibarra, A., Fernández-Castro, I & Sarasola, K., 1987, "CAPRA: An Intelligent System to Teach Novice Programmers". Perspectives in AI, vol. 2, Campbell, J. & Cuena, P. (eds.), Ellis Horwood.

Goguen, J., 1986, "Reusing and Interconnecting Software Components". Computer, Vol. 19, No. 2, Feruary 1986, pp. 16-28.

Goldberg, P.C., 1983, "The Design of Very High Level Languages", Biermann, A.W. & Guiho (eds.) *Computer Program Synthesis Methodologies*. D. Reidel Publishing Company, pp. 125-145.

Goldberg, A. & Robson, D, 1983, "Smalltalk-80: The Language and its Implementation", Addison-Wesley, Reading, Mass., 1983.

Green, C., 1969, "Application of theorem proving to Problem Solving", Proceedings of the First International Joint Conference on Artificial Intelligence, Washington, DC, pp. 219-239.

Green, C., 1977, "A Summary of the PSI Program Synthesis System", Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, Ma, August 1977, pp. 380-381.

Hammer, M., Howe, W.G., Kruskal, V.J. & Wladawsky, I., 1977, "A Very High Level Programming Language for Data Processing Applications", Communications of the ACM, Vol. 20, No. 11.

Hammond, K.J., 1989, "Case-Based Planning. Viewing Planning as a Memory Task". Academic Press, Boston, MA.

Heidorn, G.E., 1972, "Natural Language Inputs to a Simulation Programming System", Technical Report NPS-55HD72101A, Naval Postgraduate School, Monterey, California.

Horowitz, E. & Munson, J. B., 1984, "An Expansive View of Reusable Software", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 477-487.

Johnson, W.L. & Soloway, E., 1985, "PROUST: Knowledge-Based Program Understanding", IEEE Transactions on Software Engineering, Vol. SE-11, pp. 267-275.

Jones, T. C., 1984, "Reusability in Programming: A Survey of the State of the Art". IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 488-494.

Kant, E., 1985, "Understanding and Automating Algorithm Design", IEEE Transactions of Software Engineering, Vol. SE-11, No. 11, November 1985, pp. 1362-1374.

Kant, E. & Barstow, D.R., 1981, "The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis". IEEE Transactions of Software Engineering, Vol. SE-7, No. 5, September 1981, pp. 458-471.

Katz, S., Richtes, C.A. & The, K.-S., 1987, "PARIS: A System for Reusing Partially Interpreted Schemas", Proceedings of the Ninth Annual International Conference on Software Engineering, 1987. En Tracz, W. (ed.) *Tutorial: Software Reuse: Emerging Technology*, pp. 290-298, IEEE Computer Society Press, 1988.

Kernighan, B.W., 1984, "The UNIX and Software Reusability", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 513-518.

Kolodner, J.L., 1983, "Reconstructive Memory: A Computer Model", Cogitive Science, Vol. 7, pp. 281-328.

Kolodner, J.L., 1984, "Retrieval and Organizational Strategies in Conceptual Memory", Lawrence Erlbaum Associates, Hillsdale, N.J.

Kolodner, J.L., 1987, "Capitalizing on failure through case-based inference", en Proceedings of the Ninth Annual Conference of the Cognitive Science Society, pp. 715-726, Cognitive Science Society, Lawrence Erlbaum Associates, Hillsdale, N.J.

Korson, T. & McGregor, J.D., 1990, "Understanding Object-Oriented: A Unifying Paradigm", Communications of the ACM, September 1990, Vol. 33, No. 9, pp. 40-60.

Lanergan, R.G. & Grasso, C.A., 1984, "Software Engineering with Reusable Designs and Code", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 498-501.

Lebowitz, M., 1980, "Generalization and Memory in a Integrated Understanding System", Research Report 186, Yale University.

Lebowitz, M., 1983, "Generalization From Natural Language Text", Cognitive Science, Vol. 7, pp. 1-40.

Lee, R.C.T., Chang, C.L. & Waldinger, R.J., 1974, "An improved program-synthesizing algorithm and its correctness", Communications of the ACM, Vol. 17, pp. 211-217.

Manna, Z. & Waldinger, R.J., 1971, "Towards automatic program synthesis", Communications of the ACM, Vol. 14, No. 3, April 1971, pp. 151-164.

Manna, Z. & Waldinger, R.J., 1979, "Synthesis: Dreams ==> Programs", IEEE Transactions on Software Engineering, Vol. SE-5, No. 4, pp. 294- 328.

Mark, W. S. & Simpson, R.L., 1991, "Knowledge-Based Systems. An Overview", IEEE Expert, Vol. 6, No. 3, June 1991, pp. 12-17.

Matsumoto, Y., 1984, "Some Experiences in Promoting Reusable Software Presentation in Higher Abstract Levels", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 502-513.

Méndiz, I., Fernández-Chamizo, C. & Fernández-Valmayor,A., 1992, "Program Synthesis Using Case Based Reasoning", 12 IFIP Congress, Software Development and Maintenance (poster), Madrid, Septiembre 1992.

Milne, R., 1991, "Model-Based reasoning: The applications gap", IEEE Expert, Vol. 6, No. 6, December 1991, pp. 5-7.

Minsky, M., 1975, "A framework for representing knowledge". En Winston (ed.) The psycology of computer vision, pp. 211-277, New York, McGraw- Hill.

Miriyala, K. & Harandi, M. T., 1991, "Automatic Derivation of Formal Software Specifications From Informal Descriptions", IEEE Transactions on Software Engineering, Vol. SE-17, No. 10, October 1991, pp. 1126-1142.

Mitchell, T.M., Keller, R.M., & Kedar-Cabelli, S.T., 1986, "Explanation-based generalization: a unifying view", Machine Learning, Vol. 1, No. 1, 1986, pp. 47-80.

Mostow, J., 1985, "Foreword. What is AI? And What Does It Have to Do with Software Engineering?. IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November 1985, pp. 1253-1255.

Mostow, J., & Fisher, G, 1989, "Replaying Transformationals of Heuristic Search Algorithms in Diogenes", In Proceedings of the DARPA Workshop on Case-Based Reasoning, pp. 94-99, San Mateo, Calif.: Morgan Kaufmann Publishers, Inc., ??.

Newell, A., Shaw, J. & Simon, H., 1960, "Report on general problem- sloving program for a computer", Proceedings of the International Conference on Information Procesing, UNESCO, Paris.

Pratt, T.W., 1986, "Programing Languages. Design and Implementation". 2nd. ed., Prentice Hall International.

Pressman, R.S., 1987, "Software Engineering: A practitioner's approach", Mc Graw Hill.

Prieto-Díaz, R., 1991, "Implementing Faceted Classification for Software Reuse. Communications of ACM, Vol. 34, No. 5, May 1991, pp 89-97.

Prieto-Díaz, R. & Freeman, P., 1987, "Classifying Software for Reusability". IEEE Software, Vol. 4, No. 1, January 1987, pp. 6-16.

Quillian, M.R., 1967, "Word Concepts: A Theory and Simulation of Some Basic Semantic Capabilities", en Brachman, R.J. & Levesque, H.J. (eds.) *Readings in Knowledge Representation*, Morgan Kaufmann Publishers Inc., CA, 1985.

Reynols, R.G., Maletic, J.I. & Porvin, S.E., 1990, "PM: A System to Support the Automatic Acquisition of Programming Knowledge". IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 3, September 1990, pp. 273-282.

Rich, C., 1981, "A Formal Representations For Plans In The Programer's Apprentice". *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, en Rich, C. & Waters, R.C. (ed), 1986, *Readings in Artificial Intelligence and Software Engineering*, 491-506, Morgan Kaufmann Publishers, Inc., Los Altos, CA.

Rich, C. & Waters, R.C., 1986, "Readings In Artificial Intelligence and Software Engineering". William Kaufmann Publishres. Los Altos, CA.

Rich, C. & Waters, R.C., 1988, "The Programmer's Apprentice: A Research Overview". IEEE Computer, Vol. 21, No. 11, November 1988, pp. 11-23.

Rich, C. & Waters, R.C., 1988, "Automatic Programing: Myths and Prostects". IEEE Computer, Vol. 21, No. 8, August 1988, pp. 40-51.

Rich, E., 1987, "Artificial Intelligence". Encyclopedia of AI, 1987 Shapiro, S.C. editor in Chief.

Riesbeck, C.K. & Schank, R.C., 1989, "Inside Case-Based Reasoning", Lawrence Erlbaum Associates, Hillsdale, N.J.

Rissland, E.L. & Ashley, K.D., 1986, "Hypotheticals as heuristic device", en Proceedings of AAAI-86, pp. 289-297, American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc., Los Altos, CA.

Schank, R.C., 1972, "Conceptual Dependency:A Theory of Natural Language Understanding", Cognitive Psycology, 1972, pp. 552-631.

Schank, R.C., 1975, "Conceptual Information Processing", North Holland/American Elsevier, Amsterdam.

Schank, R.C., 1982, "Dynamic Memory: A Theory of Reminding and Learning in Computers and People", Cambridge University Press.

Schank, R.C., 1986, "Explanation Patterns: Understanding Mechanically and Creatively", Lawrebce Erlbaum Associates, Hillsdale, N.J.

Schank, R.C. & Abelson, R.P., 1977, "Script, Plans, Goals, and Understanding", Lawrence Erlbaum Associates, Hillsdale, N.J.

Seymour, L., 1987, "Estructura de datos", Mc Graw Hill.

Shriver, B.D., 1987, "Reuse revised", IEEE Software, Vol. 4, No. 1, January 1987, pp. 5.

Simmons, R.G., 1988, "A Theory of Debbuging", en Kolodner, J.L., Proceedings of the First Case-Based Reasoning Workshop, pp. 388-401, Morgan Kaufmann Publishers, Inc., Los Altos, CA.

Simpson, R.L., 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, School of Information and Computer Science, Atlanta GA.

Smith, C.H., 1982, "The power of pluralism for Automatic Program Synthesis", Journal of the ACM, Vol. 29, No. 4, pp. 1144-1165.

Smith, D.R., 1990, "KIDS: A Semiautomatic Program Development System", IEEE Transactiosn on Software Engineering, Vol. SE-16, No. 9, September 1990, pp 1024-1043.

Soloway, E. & Ehrlich, K., 1984, "Empirical Studies of Programming Knowledge". IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 595-609.

Steier, D ., 1991, "Automating Algorithm Design within a General Architectura for Intelligence", Automating Software Desing, AAAI Pres.

Sussman, G., 1975, "A Computer Model of Skill Acquisition", Vol. 1 of Artificial Intelligence Series, American Elsevier, New York.

Sycara, E.P., 1987, "Resolving Adversarial Conflics: An Approach to Integrating Case-Based and Analytic Methods". Technical Report GIT- ICS-87/26, Georgia Institute of Thechnology, School of Information and Computer Science, Atlanta GA.

Tracz, W., "Tutorial: Software Reuse: Emerging Technology". IEEE Computer Society Press, Washington.

de Val, A. & Morueco, J., 1989, "Diagnosis de Circuitos Electrónicos con Razonamiento Basado en Modelos", Actas de la III Reunión Técnica de la Asociación Española para la Inteligencia Artificial, Madrid, Noviembre de 1989, pp. 281-299.

Waters, R.C., 1981, "The Programer's Aprendice: A Sesion with KBEmacs". IEEE Transactions on Software Engeneering, Vol. SE-11, No. 11, November 1981, pp. 1296-1320.

Waters, R.C., 1982, "The Programmer's Aprentice: knowledge based program editing". IEEE Transactions on Software Engineering, Vol. SE- 8, No. 1, January 1982, pp 1-12.

Wiemer, R. & Sincovec, R., 1984, "Software Engineering with Modula-2 and Ada", ed. Wiley, 1984.

Wile, D.S., 1983, "Program Developments: Formal Explanations of Implementations", Communications of the ACM, Vol. 26, No. 11, November 1983, pp. 902-911.

Wirth, N., 1971, "Program Development by Stepwise Refinement", Communications of the ACM, Vol. 14, No. 4, April 1973, pp. 221-227.

Wirth, N, 1973, "Systematic Programing. An Introduction", ed. Prentice Hall, 1973.

APENDICE

```
; CBR-FBAS.LSP
; ****************************************************************
; Funcionas basicas
; ****************************************************************

; ---------------------------------
; Obtiene la FEATURE dado un CD y un ROLE

(defun FEATURE-ROLE-CD (role cd)
    (assoc role (cd-features cd)))

; ---------------------------------
; Obtiene el ROLE de una FEATURE

(defun ROLE-FEATURE (feature)
    (car feature))

; ---------------------------------
; Obtiene el FILLER (cd) de una FEATURE

(defun FILLER-FEATURE (feature)
    (cadr feature))

; ---------------------------------
; Obtiene el FILLER de un CD dado su ROLE

(defun FILLER-ROLE-CD (role cd)
    (filler-feature (feature-role-cd role cd)))

; ---------------------------------
; Obtiene el R-LINK dado un CD y un ROLE

(defun R-LINK-ROLE-CD (role cd)
    (some #'(lambda (r-link) (and (equal role
                                         (link-r-type r-link))
                                  r-link))
          (cd-r-links cd))
)
```

```
; ------------------------------------
; Devuelve la lista de los roles de los r-links cd un cd

(defun LST-ROLE-R-LINKS (cd)
   (mapcar #'(lambda (r-link) (link-r-type r-link))
                    (cd-r-links cd)))

; ------------------------------------
; Devuelve la lista de los roles de los features de un cd

(defun LST-ROLE-FEATURES (cd)
   (mapcar #'(lambda (feature) (car feature))
                    (cd-features cd)))

; ------------------------------------
; Devuelve el n-esimo cd siguiente a uno dado

(defun NTH-CD-SIG (n cd)
    (do ( (m n (- m 1))
          (cd-aux cd (link-r-to
                        (r-link-role-cd 'siguiente cd-aux))) )
        ((= m 0) cd-aux)
    )
)

; ------------------------------------
; Funcion que anade el slot enunciado a un cd

(defun PON-ENUN (cd cd-enun)
    (setf (cd-features cd)
          (append (cd-features cd)
                  (list (assoc 'enunciado
                               (cd-features cd-enun))))
    )
)

; CBR-CMPT.LSP
; ************************************************************
; Funciones de compatibilidad o variadas

; ------------------------------------
; Quita todos los elementos de una lista de otra
; 'lista-rem' lista con los elementos que se deben quitar de
; 'lista'

(defun REMOVE-LIST (lista-rem lista)
     (setq resultado lista)
```

```
      (setq remover   lista-rem)
      (loop (if (null remover) (return))
            (setq resultado (remove (car remover) resultado))
            (setq remover (cdr remover))
      )
      resultado
)


; ------------------------------------
; Funcion que quita los elementos duplicados

(defun REMOVE-DUPLICATES (l &OPTIONAL (test nil))
   (and l
        (let ((x (car l))
              (r (remove-duplicates (cdr l) test)))
             (cond ((if test (member x r :TEST test) (member x
                                                          r))
                         r)
                   (t
                         (cons x r))))
    )
)

; ------------------------------------
; Funcion que quita los elementos duplicados, pero quita por el
; final, no por el principio.

(defun REM-DUP-END (l)
    (and l
         (reverse (remove-duplicates (reverse l)))
    )
)

; ------------------------------------
; Funcion que busca en una lista, un CD igual a uno dado

(defun FIND (cd lista)
    (some #'(lambda (x) (and (iguales-cd cd x) x))
          lista)
)

; CBR-IDXC.LSP
; ****************************************************************
; Se trata de las funciones que indexan casos de entrenammiento
; ****************************************************************
```

```
; ------------------------------
; Funcion que indexa un caso. Lo que hace es ejecutar la funcion
; 'INDEXA-CD-ESPECIALIZA' para cada uno de los CDs que componen
; el caso (enunciado, pseudocodigo y codigo).

(defun INDEXA-CASO-2 (cd-enun)
    (let ((*NV* nil)
          (*NP* nil))
         (indexa-cd-especializa *TOP* cd-enun)
         (push cd-enun *NV*)
         (expandir (cd-r-links cd-enun))
         (remover-nodos-visitados *NV*)
         (loop (if (null *NP*) (return))
               (setf (cd-features (car *NP*))
                     (append (cd-features (car *NP*))
                             (list (assoc 'enunciado
                                          (cd-features cd-enun)))))
               (indexa-cd-especializa *TOP* (car *NP*))
               (push (car *NP*) *NV*)
               (expandir (cd-r-links (car *NP*)))
               (remover-nodos-visitados *NV*)
         )
    )
)

; CBR-FNSL.LSP
; ****************************************************************
; Funciones para Soluciones
; ****************************************************************

; ------------------------------
; Funcion que obtiene el programa que resuelve un enunciado

(defun SOLVER-PROBLEM (cd-enun)
    (let ((cd-siml (get-better-match cd-enun)))
         (if cd-siml
     (let ((ft-diff (get-features-diff cd-enun cd-siml)))
          (or (and (iguales-cd (filler-role-cd 'tipo cd-enun)
                               (filler-role-cd 'tipo cd-siml))
                   (or (and (iguales-cd (filler-role-cd 'de cd-enun)
                                        (filler-role-cd 'de cd-siml))
                            (ya-solver-problem cd-enun))
                       (solver-problem-ad cd-enun cd-siml ft-diff)))
              (solver-probem-aa cd-enun cd-siml ft-diff )
          )
     )
     (format t
```

```
     "~&No se ha encontrado ninguna analogia. No se puede reslver"))
    )
)

; - Cuando ya existia ese problema en la biblioteca -
(defun YA-SOLVER-PROBLEM (cd-enun)
    (format t "~&Ya existia este problema en la biblioteca")
    cd-enun
)

; - Cuando hay Analogia Directa -
(defun SOLVER-PROBLEM-AD (cd-enun cd-siml ft-diff)
    (format t "~&Se ha encontrado una analogia directa")
    (crea-link-r 'refinamiento
                 cd-enun
                 (modif-lst-seq (link-r-to (r-link-role-cd
                                            'refinamiento
                                            cd-siml))
                                ft-diff))
    cd-enun
)

; - Cuando hay Analogia Aproximada -
(defun SOLVER-PROBLEM-AA (cd-enun cd-siml ft-diff)
    (format t "~&Se ha encontrado una analogia aproximada")
)

; ------------------------------------
; Funcion que obtienen el CD que mejor empareja con uno dado de
; entrada por ahora se coje el primero que se obtiene de una
; posible lista.

(defun GET-BETTER-MATCH (cd)
    (or (loc-cd cd)
        (loc-cd (build-match-parcial cd t   nil))
        (loc-cd (build-match-parcial cd nil t))
        (loc-cd (build-match-parcial cd nil nil)))
)

; ------------------------------------
; Crea un CD que, de encontrarse en la memoria, tendria un match
; parcial con el que se desea. Hay dos variables 'oper' y 'tad'.
; Si estan a T significa que se quiere una AD (analogia Directa)
; en ese dato (operacion o estructura de dato), si estan a NIL,
; significa que solo se espera una analogia aproximada.

(defun BUILD-MATCH-PARCIAL (cd oper tad)
```

```
    (let* ((f-tip (feature-role-cd 'tipo cd))
           (f-tad (feature-role-cd 'de   cd))
           (f-res (remove f-tip (remove f-tad (cd-features cd))))

           (new-f-tip (if oper f-tip (list 'tipo (heredar-filler
                                                   cd 'clase))))
           (new-f-tad (if tad  f-tad (list 'de
                                       (selec-slots
                                          (filler-role-cd 'de cd)
                                          '(name)))))

           (new-fs (append (list new-f-tip) f-res
                                            (list new-f-tad))))
          (make-cd :HEAD (cd-head cd) :FEATURES new-fs)
    )
)

; ------------------------------
; Construye un CD de la misma cabecera y con solo los slots que
; se pasen en la lista. Si un slot de la lista no lo tuviera, no
; pone nada en su lugar

(defun SELEC-SLOTS (cd list-slots)
    (let ((feats (mapcan #'(lambda (slot)
                                  (and (feature-role-cd slot cd)
                                       (list (feature-role-cd
                                              slot cd))))
                         list-slots)))
         (make-cd :HEAD (cd-head cd) :FEATURES feats))
)


; ------------------------------
; Esta funcion devuelve un filler, es el heredado de un slot a
; lo largo de la jerarquia hasta llegar a la raiz. Podria ser que
; segun por donde se vaya, se obtiene un valor u otro. Se coge
; el primero que se encuentra

(defun HEREDAR-FILLER (cd slot)
    (or (filler-role-cd slot cd)
        (some #'(lambda (mop-padre)
                        (heredar-filler-mop mop-padre slot))
              (busca-mops-padre *top* cd))
    )
)

(defun HEREDAR-FILLER-MOP (mop slot)
```

```
    (or (filler-role-cd slot (mop-contents mop))
        (and (mop-parent mop)
             (heredar-filler-mop (mop-parent mop) slot))
    )
)


; ------------------------------------
; Localiza un CD en memoria de las carcateristicas de otro: El
; primer CD compatible.

(defun LOC-CD (cd)
    (car (mapcan #'(lambda (mop)
                        (mapcan #'(lambda (event)
                                     (and (compatible event cd)
                                          (list event)))
                                  (mop     mop)))
                   (busca-mops-padre *top* cd)
         )
    )
)


; ------------------------------------
; Funcion que obtiene las features que tienen distintas dos CDs
; dados Se devuelve una lista de features con el role y el CD
; del primer CD

(defun GET-FEATURES-DIFF (cd1 cd2)
  (mapcan #'(lambda (ft1)
      (let* ((ft2  (feature-role-cd (car ft1) cd2))
             (fll1 (cadr ft1))
             (fll2 (cadr ft2)))
            (cond ((null ft2)
                     (list ft1))
                  ((iguales-cd fll1 fll2)
                     nil)
                  ((equal (cd-head fll1) (cd-head fll2))
                     (list (list (car ft1)
                           (make-cd :HEAD (cd-head fll1)
                                    :FEATURES (get-features-diff
                                               fll1 fll2)))))
                  (t (list ft1)))
      ))
          (cd-features cd1))
)
```

```
; ------------------------------------
; Funcion que sigue un algoritmo (dado por un CD) por sus r-links
; "siguiente" y lo va modificando segun lo que marcan una sere
; de diferencias dadas por una lista de features diferentes:
; FEATURES-DIFF

(defun MODIF-LST-SEQ (cd features-diff)
   (let ((cd-solution (modif-cd cd features-diff)))
        (and (r-link-role-cd 'siguiente cd)
             (crea-link-r 'siguiente cd-solution
                          (modif-lst-seq (link-r-to
                                          (r-link-role-cd
                                          'siguiente cd))
                                         features-diff)
             ))
        cd-solution
   )
)


; ------------------------------------
; Funcion que devuelve la lista de los R-LINKS "directos" de un
; cd.
; NOTA: Un r-link "directo" es aquel que no tienen ''

(defun R-LINKS-DIRECTAS (cd)
   (mapcan #'(lambda (r-link)
             (and (/= 60 (char (symbol-name (link-r-type r-link))
                  0))
                  (list r-link)))
           (cd-r-links cd))
)


; ------------------------------------
; Funcion que devuleve un CD igual a otro, pero sin el slot
; 'enunciado' y sin la r-link 'siguiente en el CD y en todos
; los relacionados con el por las r-links

(defun SEMI-COPIAR (cd &optional (ft-rem '(enunciado))
       (rl-rem '(siguiente)))
  (let* ((feat-rem (mapcan #'(lambda (role)
                               (and (feature-role-cd role cd)
                                    (list (feature-role-cd role
                                          cd))))
                           ft-rem))
         (rlin-rem (mapcan #'(lambda (role)
```

```
                              (and (r-link-role-cd role cd)
                                   (list (r-link-role-cd role
                                         cd))))
                              rl-rem))
          (cd-aux
           (make-cd
              :HEAD (cd-head cd)
              :FEATURES (remove-list feat-rem
                        (cd-features cd))
           )))
           (mapcar #'(lambda (r-link)
                (crea-link-r (link-r-type r-link)
                             cd-aux
                             (semi-copiar (link-r-to r-link)
                                          ft-rem rl-rem)))
                (remove-list rlin-rem (r-links-directas cd)))
        cd-aux
  )
)

; ------------------------------------
; Esta funcion, dado un CD y una lista de FEATURES, crea otro CD
; igual al anterior, pero si alguna de sus features estan en
; FEATURES, se pone la de FEATURES

(defun CHANGE-FEATURES (cd list-features)
     (let ((features (mapcar #'(lambda (f) (or (assoc (car f)
                                                      list-features)
                                               f))
                             (cd-features cd))))
          (make-cd :HEAD     (cd-head cd)
                   :FEATURES features)
    )
)


; ------------------------------------
; Funcion que modifica un CD de tipo secuencia
; El proceso seria el siguiente:
;    - Se localiza en meoria un CD como el de entrada, pero con
;      las diferencias cambiadas.
;      a) Si existe, se toma ese cd con sus r-links (excepto
;         'siguiente) y sus features, exceto 'enunciado. Si ese
;         cd tuviera el r-link 'refinamiento, entonces, si que se
;         tomarian los siguientes de sus "refinados"
;      b) Si no existe, se mira si tiene un r-link refinamiento
;         1. Si tiene rlink refinamiento, se cominenza de nuevo
```

```
;            como si fuera un algoritmo.
;         2. Si no lo tiene, se devuelve el CD de entrada, son
;            sus r-links execpto 'siguiente y sus features excepto
;            'enunciado


(defun MODIF-CD-SEQ (cd features-diff)
    (let ((cd-loc (loc-cd (change-features cd features-diff))))
         (cond (cd-loc
                  (cond ((r-link-role-cd 'refinamiento cd-loc)
                         (let ((cd-loc2
                                 (semi-copiar cd-loc
                                      '(enunciado)
                                      '(siguiente refinamiento))))
                              (crea-link-r 'refinamiento
                                     cd-loc2
                                     (semi-copiar
                  (link-r-to (r-link-role-cd 'refinamiento cd-loc))
                                       '(enunciado) nil ))
                              cd-loc2))
                         (T    (semi-copiar cd-loc))))

         (T
                  (cond ((r-link-role-cd 'refinamiento cd)
                         (let ((cd-2 (semi-copiar cd
                                     '(enunciado)
                                     '(siguiente refinamiento))))
                              (crea-link-r 'refinamiento
                                  cd-2
                                  (modif-lst-seq
                                     (link-r-to
                                     (r-link-role-cd 'refinamiento
                                                     cd))
                                     features-diff))
                              cd-2))
                    (t (semi-copiar cd))
                )
         )
       )
    )
)

; ------------------------------------
; Funcion que modifica un CD.

(defun MODIF-CD (cd features-diff)
     (cond ((iguales-cd (filler-role-cd 'tipo cd)
```

```
                    (list-cd '(PREGUNTAR)))
               (modif-lst-cond cd features-diff))
          ((iguales-cd (filler-role-cd 'tipo cd)
                    (list-cd '(SELECTIVA)))
               (modif-cd-no-seq cd features-diff))
          ((iguales-cd (filler-role-cd 'tipo cd)
                    (list-cd '(REPETITIVA)))
               (modif-cd-no-seq cd features-diff))
          (T
               (modif-cd-seq cd features-diff))
     )
)

; ------------------------------------
; Funcion que nodifica un CD no SECUENCIA

(defun MODIF-CD-NO-SEQ (cd features-diff)
     (let ((rlinks (mapcar #'(lambda (rl) (link-r-type rl))
          (cd-r-links cd))))
          (let ((cd-aux (semi-copiar cd '(enunciado) rlinks)))
               (mapcar #'(lambda (rl)
                         (crea-link-r (link-r-type rl)
                                 cd-aux
                                 (modif-lst-seq
                                 (link-r-to rl)
                                 features-diff)))
                    (r-links-directas cd))
               cd-aux)
     )
)


; ------------------------------------
; Funcion que modifica una lista de condiciones

(defun MODIF-LST-COND (cd features-diff)
    (let ((cd-solution (modif-cd-cond cd features-diff)))
         (or (and (r-link-role-cd 'y-logico cd)
                  (crea-link-r 'y-logico cd-solution
                               (modif-lst-cond
                                  (link-r-to
                                     (r-link-role-cd 'y-logico cd))
                                  features-diff)))
             (and (r-link-role-cd 'o-logico cd)
                  (crea-link-r 'o-logico cd-solution
                               (modif-lst-cond
                                  (link-r-to
```

```
                                    (r-link-role-cd 'o-logico cd))
                                features-diff))))
        cd-solution)
)


; ------------------------------
; Funcion que modifica un CD CONDICION

(defun MODIF-CD-COND (cd features-diff)
    (let ((cd-loc (loc-cd (change-features cd features-diff))))
        (cond (cd-loc  (semi-copiar cd-loc))
              (t (cond ((r-link-role-cd 'refinamiento cd)
                        (let ((cd-2 (semi-copiar cd
                                                 '(enunciado)
                                                 '(refinamiento))))
                             (crea-link-r 'refinamiento
                                          cd-2
                    (modif-lst-cond
                    (link-r-to (r-link-role-cd 'refinamiento cd))
                                          features-diff))
                          cd-2))
                    (t  (semi-copiar cd))
                ))
        ))
)

 ; CBR-CASO.LSP

; ================================================================
;  Crear la red CONTAR LES
; ================================================================

(setq C1-E (list-cd '(ACTION ((tipo   CONTAR)
                              (objeto NUM-NODOS)
                              (de     (PP ((tipo  ESTRUC-DATOS)
                                           (name  LISTA)
                                           (imple ENLAZADA)
                                           (forma SIMPLE))))
                              (long   INDETERMINADA)))))

; === Algoritmo ===

; - es lista vacia ?

(setq C1-A      (list-cd '(ACTION ((tipo SELECTIVA)))))
```

```
(setq C1-AC     (list-cd '(ACTION ((tipo   PREGUNTAR)
                                   (objeto LONGITUD)
                                   (de     (PP ((tipo  ESTRUC-DATOS)
                                                (name  LISTA)
                                                (imple ENLAZADA)
                                                (forma SIMPLE))))
                                   (es      NULA)))))

(setq C1-ACr    (list-cd '(ACTION ((tipo     COMPARAR)
                                   (operador IGUALDAD)
                                   (exp1 LISTA)
                                   (exp2 NIL)))))

; - Si lista vacia

(setq C1-AT (list-cd '(ACTION ((tipo   CONTAR)
                               (objeto NUM-NODOS)
                               (de     (PP ((tipo  ESTRUC-DATOS)
                                            (name  LISTA)
                                            (imple ENLAZADA)
                                            (forma SIMPLE))))
                               (long   NULA)))))

(setq C1-ATr    (list-cd '(ACTION ((tipo   DEVOLVER)
                                   (objeto CERO)))))

(setq C1-ATrr   (list-cd '(ACTION ((tipo   ASIGNAR)
                                   (var    CONTAR)
                                   (val    0)))))


; - Si lista no vacia

(setq C1-AF (list-cd '(ACTION ((tipo   CONTAR)
                               (objeto NUM-NODOS)
                               (de     (PP ((tipo  ESTRUC-DATOS)
                                            (name  LISTA)
                                            (imple ENLAZADA)
                                            (forma SIMPLE)
                                            (forma SIMPLE))))
                               (long   NO-NULA)))))

(setq C1-AFr1   (list-cd '(ACTION ((tipo   INICIALIZAR)
                                   (objeto CONTADOR)
                                   (valor  UNO)))))

(setq C1-AFr1r  (list-cd '(ACTION  ((tipo   ASIGNAR)
```

```
                                    (var    CONTADOR)
                                    (val    1)))))

(setq C1-AFr2   (list-cd '(ACTION ((tipo   APUNTAR)
                                   (objeto PUNTERO)
                                   (a      PRIM-ELEM)
                                (de     (PP ((tipo  ESTRUC-DATOS)
                                             (name  LISTA)
                                             (imple ENLAZADA)
                                             (forma SIMPLE)))))
                          )
))

(setq C1-AFr2r  (list-cd '(ACTION ((tipo   ASIGNAR)
                                   (var    PUNTERO)
                                   (val    LISTA)))))

(setq C1-AFr3   (list-cd '(ACTION ((tipo   REPETITIVA)))))

(setq C1-AFr30  (list-cd '(ACTION ((tipo   PREGUNTAR)
                                   (objeto ESTE-ELEM)
                                (de     (PP ((tipo  ESTRUC-DATOS)
                                             (name  LISTA)
                                             (imple ENLAZADA)
                                             (forma SIMPLE))))
                                   (noes   ULTIMO)))))


(setq C1-AFr31  (list-cd '(ACTION ((tipo     INCREMENTAR)
                                   (objeto   CONTADOR)
                                   (cantidad UNO)))))

(setq C1-AFr31r (list-cd '(ACTION ((tipo   ASIGNAR)
                                   (var    CONTADOR)
                          (val    (ACTION ((tipo     OPERAR-ARIT)
                                           (operador ADICION)
                                           (exp1     CONTADOR)
                                           (exp2     1)))))))

(setq C1-AFr32  (list-cd '(ACTION ((tipo   APUNTAR)
                                   (objeto PUNTERO)
                                   (a      SIG-ELEM)
                                (de     (PP ((tipo  ESTRUC-DATOS)
                                             (name  LISTA)
                                             (imple ENLAZADA)
                                           (forma SIMPLE)))))
                          )
```

```
))

(setq C1-AFr4   (list-cd '(ACTION ((tipo   DEVOLVER)
                                   (objeto CONTADOR)))))

(setq C1-AFr4r  (list-cd '(ACTION ((tipo   ASIGNAR)
                                   (var    CONTAR)
                                   (val    CONTADOR)))))

; -- Relaciones

(crea-link-r 'refinamienro C1-E C1-A)

(crea-link-r 'condicion   C1-A C1-AC)
(crea-link-r 'siguiente-T C1-A C1-AT)
(crea-link-r 'siguiente-F C1-A C1-AF)

(crea-link-r 'refinamiento C1-AT C1-ATr)
(crea-link-r 'refinamiento C1-AF C1-AFr1)

(crea-link-r 'siguiente C1-AFr1 C1-AFr2)
(crea-link-r 'siguiente C1-AFr2 C1-AFr3)
      (crea-link-r 'condicion C1-AFr3 C1-AFr30)
      (crea-link-r 'contiene  C1-AFr3 C1-AFr31)
      (crea-link-r 'siguiente   C1-AFr31 C1-AFr32)
(crea-link-r 'siguiente C1-AFr3 C1-AFr4)

(crea-link-r 'refinamiento C1-AC    C1-ACr)
(crea-link-r 'refinamiento C1-ATr   C1-ATrr)
(crea-link-r 'refinamiento C1-AFr1  C1-AFr1r)
(crea-link-r 'refinamiento C1-AFr2  C1-AFr2r)
(crea-link-r 'refinamiento C1-AFr30 C1-AFr30r)
(crea-link-r 'refinamiento C1-AFr31 C1-AFr31r)
(crea-link-r 'refinamiento C1-AFr32 C1-AFr32r)
(crea-link-r 'refinamiento C1-AFr4  C1-AFr4r)

(format t "~&Creado el caso C1")


; ============================================================
;  Crear la red CONTAR LEC
; ============================================================

(setq C2-E (list-cd '(ACTION ((tipo   CONTAR)
                              (objeto NUM-NODOS)
                              (de     (PP ((tipo  ESTRUC-DATOS)
                                           (name  LISTA)
```

```
                                        (imple ENLAZADA)
                                        (forma CIRCULAR))))
                                (long   INDETERMINADA)))))

; === Algoritmo ===

; - es lista vacia ?

(setq C2-A      (list-cd '(ACTION ((tipo SELECTIVA)))))

(setq C2-AC     (list-cd '(ACTION ((tipo   PREGUNTAR)
                                (objeto LONGITUD)
                                (de     (PP ((tipo  ESTRUC-DATOS)
                                             (name  LISTA)
                                             (imple ENLAZADA)
                                             (forma CIRCULAR))))
                                    (es      NULA)))))

(setq C2-ACr    (list-cd '(ACTION ((tipo     COMPARAR)
                                   (operador IGUALDAD)
                                   (exp1 LISTA)
                                   (exp2 NIL)))))

; - Si lista vacia

(setq C2-AT (list-cd '(ACTION ((tipo   CONTAR)
                               (objeto NUM-NODOS)
                               (de     (PP ((tipo  ESTRUC-DATOS)
                                            (name  LISTA)
                                            (imple ENLAZADA)
                                            (forma CIRCULAR))))
                               (long   NULA)))))

(setq C2-ATr    (list-cd '(ACTION ((tipo   DEVOLVER)
                                   (objeto CERO)))))

(setq C2-ATrr   (list-cd '(ACTION ((tipo   ASIGNAR)
                                   (var    CONTAR)
                                   (val    0)))))


; - Si lista no vacia

(setq C2-AF (list-cd '(ACTION ((tipo   CONTAR)
                               (objeto NUM-NODOS)
                               (de     (PP ((tipo  ESTRUC-DATOS)
                                            (name  LISTA)
```

```
                                        (imple ENLAZADA)
                                        (forma CIRCULAR)
                                        (forma CIRCULAR))))
                                (long   NO-NULA)))))

(setq C2-AFr1   (list-cd '(ACTION ((tipo   INICIALIZAR)
                                   (objeto CONTADOR)
                                   (valor  UNO)))))

(setq C2-AFr1r  (list-cd '(ACTION  ((tipo   ASIGNAR)
                                    (var    CONTADOR)
                                    (val    1)))))

(setq C2-AFr2   (list-cd '(ACTION ((tipo   APUNTAR)
                                   (objeto PUNTERO)
                                   (a      PRIM-ELEM)
                                   (de     (PP ((tipo  ESTRUC-DATOS)
                                                (name  LISTA)
                                                (imple ENLAZADA)
                                                (forma CIRCULAR))))))
                            )
))

(setq C2-AFr2r  (list-cd '(ACTION ((tipo   ASIGNAR)
                                   (var    PUNTERO)
                                   (val    LISTA)))))

(setq C2-AFr3   (list-cd '(ACTION ((tipo   REPETITIVA)))))

(setq C2-AFr30  (list-cd '(ACTION ((tipo   PREGUNTAR)
                                   (objeto ESTE-ELEM)
                                   (de     (PP ((tipo  ESTRUC-DATOS)
                                                (name  LISTA)
                                                (imple ENLAZADA)
                                                (forma CIRCULAR))))
                                   (noes   ULTIMO)))))


(setq C2-AFr30r (list-cd '(ACTION ((tipo     COMPARAR)
                                   (operador DESIGUALDAD)
                                   (exp1     (ACTION
                                             ((tipo DESIG-CAMPO)
                                              (var (ACTION ((tipo REF-VAR)
                                                            (var PUNTERO))))
                                              (cam SIG))))
                                   (exp2   LISTA)))))
```

```
(setq C2-AFr31  (list-cd '(ACTION ((tipo     INCREMENTAR)
                                   (objeto   CONTADOR)
                                   (cantidad UNO)))))

(setq C2-AFr31r (list-cd '(ACTION ((tipo   ASIGNAR)
                                   (var    CONTADOR)
                                   (val    (ACTION ((tipo     OPERAR-ARIT)
                                                    (operador ADICION)
                                                    (exp1     CONTADOR)
                                                    (exp2     1)))))))))

(setq C2-AFr32  (list-cd '(ACTION ((tipo   APUNTAR)
                                   (objeto PUNTERO)
                                   (a      SIG-ELEM)
                                   (de     (PP ((tipo  ESTRUC-DATOS)
                                                (name  LISTA)
                                                (imple ENLAZADA)
                                                (forma CIRCULAR)))))
                           )
))

(setq C2-AFr32r (list-cd '(ACTION ((tipo ASIGNAR)
                                   (var  PUNTERO)
                                   (val  (ACTION
                                           ((tipo DESIG-CAMPO)
                                            (var (ACTION ((tipo REF-VAR)
                                                          (var PUNTERO))))
                                            (cam SIG)))))
                           )
))

(setq C2-AFr4   (list-cd '(ACTION ((tipo   DEVOLVER)
                                   (objeto CONTADOR)))))

(setq C2-AFr4r  (list-cd '(ACTION ((tipo   ASIGNAR)
                                   (var    CONTAR)
                                   (val    CONTADOR)))))

; -- Relaciones

(crea-link-r 'refinamiento C2-E C2-A)       ; Link Enunciado - refinamiento

(crea-link-r 'condicion   C2-A C2-AC)
(crea-link-r 'siguiente-T C2-A C2-AT)
(crea-link-r 'siguiente-F C2-A C2-AF)

(crea-link-r 'refinamiento C2-AT C2-ATr)
```

```
(crea-link-r 'refinamiento C2-AF C2-AFr1)

(crea-link-r 'siguiente C2-AFr1 C2-AFr2)
(crea-link-r 'siguiente C2-AFr2 C2-AFr3)
      (crea-link-r 'condicion C2-AFr3 C2-AFr30)
      (crea-link-r 'contiene  C2-AFr3 C2-AFr31)
      (crea-link-r 'siguiente   C2-AFr31 C2-AFr32)
(crea-link-r 'siguiente C2-AFr3 C2-AFr4)

(crea-link-r 'refinamiento C2-AC    C2-ACr)
(crea-link-r 'refinamiento C2-ATr   C2-ATrr)
(crea-link-r 'refinamiento C2-AFr1  C2-AFr1r)
(crea-link-r 'refinamiento C2-AFr2  C2-AFr2r)
(crea-link-r 'refinamiento C2-AFr30 C2-AFr30r)
(crea-link-r 'refinamiento C2-AFr31 C2-AFr31r)
(crea-link-r 'refinamiento C2-AFr32 C2-AFr32r)
(crea-link-r 'refinamiento C2-AFr4  C2-AFr4r)

(format t "~&Creado el caso C2")


;
;============================================================================
;  Crear la red BUSCAR LES
;
;============================================================================

(setq B1-E (list-cd '(ACTION ((tipo   BUSCAR)
                              (objeto ELEMENTO)
                              (de     (PP ((tipo  ESTRUC-DATOS)
                                           (name  LISTA)
                                           (imple ENLAZADA)
                                           (forma SIMPLE))))
                              (long   INDETERMINADA)))))

; === Algoritmo ===

; - es lista vacia ?

(setq B1-A      (list-cd '(ACTION ((tipo SELECTIVA)))))

(setq B1-AC     (list-cd '(ACTION ((tipo   PREGUNTAR)
                                   (objeto LONGITUD)
                                   (de     (PP ((tipo  ESTRUC-DATOS)
                                                (name  LISTA)
                                                (imple ENLAZADA)
                                                (forma SIMPLE))))
```

```
                                    (es      NULA)))))

(setq B1-ACr     (list-cd '(ACTION ((tipo     COMPARAR)
                                    (operador IGUALDAD)
                                    (exp1 LISTA)
                                    (exp2 NIL)))))

; - Si lista vacia

(setq B1-AT (list-cd '(ACTION ((tipo    BUSCAR)
                               (objeto  ELEMENTO)
                               (de      (PP ((tipo  ESTRUC-DATOS)
                                             (name  LISTA)
                                             (imple ENLAZADA)
                                             (forma SIMPLE))))
                               (long    NULA)))))

(setq B1-ATr     (list-cd '(ACTION ((tipo   DEVOLVER)
                                    (objeto NIL)))))

(setq B1-ATrr    (list-cd '(ACTION ((tipo   ASIGNAR)
                                    (var    BUSCAR)
                                    (val    NIL)))))


; - Si lista no vacia

(setq B1-AF (list-cd '(ACTION ((tipo   BUSCAR)
                               (objeto  ELEMENTO)
                               (de      (PP ((tipo  ESTRUC-DATOS)
                                             (name  LISTA)
                                             (imple ENLAZADA)
                                             (forma SIMPLE))))
                               (long    NO-NULA)))))

(setq B1-AFr1    (list-cd '(ACTION ((tipo   APUNTAR)
                                    (objeto PUNTERO)
                                    (a      PRIM-ELEM)
                                    (de     (PP ((tipo  ESTRUC-DATOS)
                                                 (name  LISTA)
                                                 (imple ENLAZADA)
                                                 (forma SIMPLE)))))
                          )
))

(setq B1-AFr1r   (list-cd '(ACTION ((tipo   ASIGNAR)
                                    (var    PUNTERO)
```

```
                                    (val    LISTA)))))

(setq B1-AFr2   (list-cd '(ACTION ((tipo   REPETITIVA)))))

(setq B1-AFr2C1 (list-cd '(ACTION ((tipo   PREGUNTAR)
                                    (objeto ESTE-ELEM)
                                    (de     (PP ((tipo  ESTRUC-DATOS)
                                                 (name  LISTA)
                                                 (imple ENLAZADA)
                                                 (forma SIMPLE))))
                                    (noes   ULTIMO)))))

(setq B1-AFr2C1r (list-cd '(ACTION ((tipo     COMPARAR)
                                     (operador DESIGUALDAD)
                                     (exp1     (ACTION
                                          ((tipo DESIG-CAMPO)
                                           (var (ACTION ((tipo REF-VAR)
                                                         (var PUNTERO))))
                                           (cam SIG))))
                                     (exp2   NIL)))))

(setq B1-AFr2C2 (list-cd '(ACTION ((tipo   PREGUNTAR)
                                    (objeto ESTE-ELEM)
                                    (de     (PP ((tipo  ESTRUC-DATOS)
                                                 (name  LISTA)
                                                 (imple ENLAZADA)
                                                 (forma SIMPLE))))
                                    (noes   ELEMENTO)))))

(setq B1-AFr2C2r (list-cd '(ACTION ((tipo     COMPARAR)
                                     (operador DESIGUALDAD)
                                     (exp1     (ACTION
                                          ((tipo DESIG-CAMPO)
                                           (var (ACTION ((tipo REF-VAR)
                                                         (var PUNTERO))))
                                           (cam INFO))))
                                     (exp2      ELEMENTO)))))

(setq B1-AFr21  (list-cd '(ACTION ((tipo   APUNTAR)
                                    (objeto PUNTERO)
                                    (a      SIG-ELEM)
                                    (de     (PP ((tipo  ESTRUC-DATOS)
                                                 (name  LISTA)
                                                 (imple ENLAZADA)
                                                 (forma SIMPLE)))))

                          )
))
```

```
(setq B1-AFr21r (list-cd '(ACTION ((tipo ASIGNAR)
                                   (var  PUNTERO)
                                   (val  (ACTION
                                           ((tipo DESIG-CAMPO)
                                            (var (ACTION ((tipo REF-VAR)
                                                          (var PUNTERO))))
                                            (cam SIG)))))
                                )
))

(setq B1-AFr3   (list-cd '(ACTION ((tipo   DEVOLVER)
                                   (objeto PUNTERO)))))

(setq B1-AFr3r  (list-cd '(ACTION ((tipo   ASIGNAR)
                                   (var    BUSCAR)
                                   (val    PUNTERO)))))

; -- Relaciones

(crea-link-r 'refinamiento B1-E B1-A)       ; Link Enunciado - refinamiento

(crea-link-r 'condicion   B1-A B1-AC)
(crea-link-r 'siguiente-T B1-A B1-AT)
(crea-link-r 'siguiente-F B1-A B1-AF)

(crea-link-r 'refinamiento B1-AT B1-ATr)
(crea-link-r 'refinamiento B1-AF B1-AFr1)

(crea-link-r 'siguiente B1-AFr1 B1-AFr2)
      (crea-link-r 'condicion B1-AFr2   B1-AFr2C1)
      (crea-link-r 'y-logico  B1-AFr2C1 B1-AFr2C2)
      (crea-link-r 'contiene  B1-AFr2   B1-AFr21)
(crea-link-r 'siguiente B1-AFr2 B1-AFr3)

(crea-link-r 'refinamiento B1-AC     B1-ACr)
(crea-link-r 'refinamiento B1-ATr    B1-ATrr)
(crea-link-r 'refinamiento B1-AFr1   B1-AFr1r)
(crea-link-r 'refinamiento B1-AFr2C1 B1-AFr2C1r)
(crea-link-r 'refinamiento B1-AFr2C2 B1-AFr2C2r)
(crea-link-r 'refinamiento B1-AFr3   B1-AFr3r)

(format t "~&Creado el caso B1")

; CBR-OPER.LSP
; ============================================================
; Definiciones de operaciones en el diccionario de conceptos
```

```
; ============================================================

;
; == CONTAR

(setq AccC (list-cd '(ACTION ((clase        RECORRER)
                              (ini_cla      (ACCION ((tipo   APUNTAR)
                                                     (objeto PUNTERO)
                                                     (a      PRIM-ELEM)
                                                     (de     PP))))
                              (cd_item_cla (ACCION ((tipo   APUNTAR)
                                                     (objeto PUNTERO)
                                                     (a      SIG-ELEM)
                                                     (de     PP))))
                              (tipo         CONTAR)
                              (ini_opr      (ACCION ((tipo   INICILIZAR)
                                                     (objeto CONTADOR)
                                                     (valor  UNO))))
                             (cd_item_opr (ACCION ((tipo   INCREMENTAR)
                                                     (objeto CONTADOR)
                                                     (cantidad  UNO))))
                             (resultado   (ACCION ((tipo   DEVOLVER)
                                                     (objeto CONTADOR)))))
)))

;
; == BUSCAR

(setq AccC (list-cd '(ACTION ((clase        RECORRER)
                               (ini_cla      (ACCION ((tipo   APUNTAR)
                                                      (objeto PUNTERO)
                                                      (a      PRIM-ELEM)
                                                      (de     PP))))
                               (cd_item_cla (ACCION ((tipo   APUNTAR)
                                                      (objeto PUNTERO)
                                                      (a      SIG-ELEM)
                                                      (de     PP))))
                                (tipo         BUSCAR)
                                (cd_item_opr (ACCION ((tipo   PREGUNTAR)
                                                      (objeto ESTE-ELEM)
                                                      (de     PP)
                                                      (noes   ELEMENTO))))
                                (resultado   (ACCION ((tipo   DEVOLVER)
                                                      (objeto PUNTERO)))))
)))

;
```