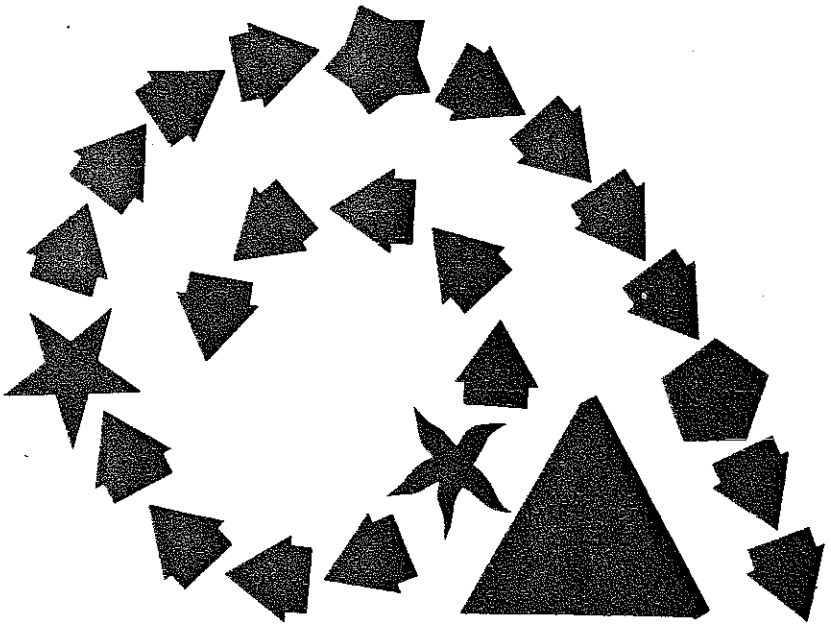
III REUNION TECNICA
DE LA ASOCIACION ESPAÑOLA
PARA LA INTELIGENCIA ARTIFICIAL

aepia 89



ACTAS

Madrid, 15, 16 y 17 de noviembre de 1989

OPERADORES DE CONSTRUCCIÓN DE ESTRATEGIAS EN SISTEMAS EXPERTOS

J. Agustí Cullell, C. Sierra, J. Pluss (*), R. Toledo (*)

Centre d'Estudis Avançats de Blanes, CSIC
17300 Blanes, Girona
tf: 972 336101
(*) Dpto de Sistemas e Informática
Fac. de Ciencias Exactas e Ingeniería
UNR, Conicet
2000 Rosario, Argentina

Palabras Clave: Lenguajes, Arquitecturas.

RESUMEN

En esta publicación presentamos un conjunto de operadores para la construcción de estrategias de resolución de problemas mediante operadores sobre módulos en lenguajes modulares. La primera parte muestra una visión de la modularización de lenguajes basados en reglas como una aplicación funcional de la teoría de módulos actualmente implementada utilizando MILORD [Sierra, 89] (lenguaje para la construcción de sistemas expertos desarrollado en el CEAB) como lenguaje base sobre el que se aplica dicha teoría. A continuación extendemos el concepto de módulos a las estrategias y definimos un conjunto de operadores que permiten la formulación de las mismas haciendo uso de las ventajas que brinda la modularización. Cabe agregar que aunque la teoría es practicamente independiente del lenguaje de reglas de base que se utilice, el presente trabajo se ha desarrollado en su totalidad sobre el sistema basado en reglas MILORD.

ABSTRACT

In this paper we are presenting a set of operators for the construction of problem-solving strategies through operators upon modules in modular languages. First we show a survey of the modularization of rule based languages as a functional aplication of the modular theory that is nowadays being implemented using MILORD [Sierra, 89] (a language for the construction of Expert Systems developed in CEAB) as the base language upon which the aforementioned theory is aplied. Then we extend the concept of modules to the strategies and define a set of operators that allow their formulation taking advantage of the facilities provided by modularization. Furthermore, we must state that although theory is almost independent from the rule based language we use, this paper has been totally developed upon the rule based system MILORD.

Este trabajo ha sido financiado parcialmente por un convenio de colaboración Hispano-Argentino y por el proyecto SPES (CICYT, 880J382)

1.- INTRODUCCIÓN

Actualmente la mayoría de los lenguajes basados en reglas carecen de primitivas que permitan una buena estructuración del conocimiento. A lo sumo, se presentan casos de modulos no estructurados, como por ejemplo los rule-sets en KEE.
Una buena estructuración de los sistemas basados en conocimientos, solucionaría los graves problemas actuales de desarrollo y mantenimiento de grandes aplicaciones. El objetivo del presente trabajo es mostrar que, aunque la experiencia se este realizando sobre MILORD, la técnica de modularización y el uso de estrategias se pueden aplicar a cualquier lenguaje basado en reglas con un costo mínimo.
Teniendo en cuenta esto último se proponen estructuras de modulos independientes del lenguaje de base, lo cual nos permite conservar todas o casi todas las caracteristicas del mismo, tales como el tipo de motores, el tipo de lógica, etc.
La clave para lograrlo reside en que nuestra aproximación no interpreta semánticamente los módulos como ocurre en [Sticklen et al 87] sino que desde nuestra aproximación, un módulo es considerado como una abstracción funcional. Esta abstracción viene representada mediante una interfaz de importación y otra de exportación.
La propuesta esta basada en el lenguaje modular ML [Harper et al 86] Otros trabajos han utilizado aproximaciones similares en la modularización de lenguajes lógicos y funcionales [O'Keefe 85], [Miller 86], [Shiver y Wegner 87].
Una vez lograda la estructura modular, el paso siguiente es la implementación de estrategias dentro de la misma para lograr una estructuración lo más completa posible de los procesos de resolución de problemas.
Por último se proponen operadores que faciliten la formulación de estrategias y en lo posible incrementen la eficiencia de ejecución de las bases de conocimiento. A lo largo de la publicación veremos una explicación detallada de la teoría de modularización propuesta, aplicada sobre MILORD ,con ejemplos tomados de la aplicación Pneumon-IA. [Verdaguer, 89].

2. MODULARIZACIÓN EN MILORD

El mecanismo de modularización que presentaremos permitirá definir especificaciones que representarán los componentes de visibilidad, interfaz de importación y exportación de los módulos, y que permitirán comprobaciones de tipos. Al mismo tiempo, se definirán módulos paramétricos que doten al sistema de una mayor potencia de representación.
El mayor interés de la técnica que propondremos es su reutilización sobre otros lenguajes basados en reglas con un costo muy bajo. Debido a esto imponemos la restricción de separar el lenguaje de modularización del lenguaje de reglas. Esta visión estática de los módulos hace que los algoritmos de evaluación del lenguaje de base no hayan de ser modificados. Los motores, los sistemas de mantenimiento de la verdad, etc., de un lenguaje que incorporara módulos dinámicos, creados en tiempos de ejecución, tendrían que ser modificados substancialmente.
Entre las motivaciones para modularizar las bases de conocimientos podemos destacar:

1) reducir la dificultad de diseño, verificación y mantenimiento de las BC.
2) disponer de espacio de nombres distintos para cada módulo, de forma que las interacciones por efectos laterales desaparezcan.

A lo largo de este capítulo utilizaremos un ejemplo, que hará de hilo conductor de la explicación, extraido del dominio de aplicación de PNEUMON-IA. La aplicación PNEUMON-IA no ha utilizado en su desarrollo las primitivas de modularización, dado que estas han sido incorporadas a MILORD posteriormente a su desarrollo.

2.1. Sintaxis del lenguaje de módulos

Aqui presentamos la sintaxis del nivel estructural utilizando módulos y especificaciones.

programa ::= **Especificación** *vinclespec* |
Modulo *vinclemod* |
Conjunto *vinclecony* |
programa programa

{**Heredar** *id* es equivalente a **modulo** *id* = *id*}

vinclespec ::= *idespec* = *exprespec*
vinclemod ::= *idmod* [(*listaparam*)]: *exprespec* | = *expremod*
listaparam ::= *idparam* : *exprespec* |**compartir** *eqcamino*|; *listaparam* |λ
eqcamino ::= *id*1 = . . . = *idn* n > 1

{*idmod* |(*listaparam*)| : *exprespec* = *expremod es equivalente a*
idmod |(*listaparam*)| = *expremod* : *exprespec*}

vinclecony ::= *idconjunto* = (*hecho*$_1$: *tipo*$_1$; . . . ; *hecho*$_n$: *tipo*$_n$) |
idconjunto = *idconjunto operador-conjunto idconjunto*

operador-conjunto ::= ∪ | ∩ | /
exprespec ::= *idespec* | **inicio** *espec* **final**
espec ::= **Modulo** *idmod* : *exprespec* |
importados |
exportados |
espec espec

expremod ::= *idcami* |(*expremod*$_1$; . . . ; *expremod*$_n$)| |
inicio *decl* **final** |
expremod : *exprespec*
idcami ::= *idmod* | *idmod* -> *idcami*
decl ::= |*cabecera*| *cuerpo*
cabecera ::= *declnucleo* | *declmod* | *declmod declnucleo*
declnucleo ::= **Abrir** *idmod* | *importados* | *exportados* |
declnucleo declnucleo
importados ::= **importa** = (*hecho*$_1$: *tipo*$_1$;. . . ;*hecho*$_n$: *tipo*$_n$) |
importa = *idconjunto*
exportados ::= **Exporta** = (*hecho*$_1$: *tipo*$_1$;. . . ;*hecho*$_n$: *tipo*$_n$) |
Exporta = *idconjunto*
cuerpo ::= *precondicion reglas metareglas*
reglas ::= **Reglas** *lreglas* | λ*
metareglas ::= **Metareglas** *lmre* | λ
premisa ::= *premisa* **y** *condición* | *condición*
lreglas ::= *regla lreglas* | *regla*
lmre ::= *mre lmre* | λ

Regla, *mre*, *condición*, son elementos no concretados en esta sintaxis. Pueden ser sustituidos por la sintaxis del lenguaje de reglas que se desee modularizar. En los siguientes apartados explicaremos los elementos más destacados de la sintaxis.

* λ representa la cadena vacia

2.1.1 Declaraciones primitivas

Los hechos que hayan de ser visibles por parte del modulo tendran que ser declarados explícitamente, así como los hechos que el resto del programa podrá ver del modulo en curso. Este mecanismo permite esconder información ("information hiding").
Los hechos afirmados por el usuario en tiempo de ejecución son hechos que no son exportados por ningun módulo del programa y reciben un tratamiento distinto al del resto.

Importa $hecho_1, \ldots hecho_n$

Indica que los hechos $hecho_1, \ldots hecho_n$ serán visibles dentro del módulo y por tanto podran formar parte de premisas de reglas y metareglas y su valor será obtenido en tiempo de ejecución. Son hechos que no son exportados por ningún módulo sinó que son afirmados por el usuario. Los hechos declarados como **Importa** no son exportados fuera del módulo. Si se quisiera exportar alguno de los hechos importados se tendria que incluir también en la lista de hechos **Exporta**. Obligamos a declarar los hechos que utilizaremos como una disciplina del programador. Los hechos que realmente se utilizan tendrán que ser un subconjunto de los que se dice que se utilizarán.

Exporta $hecho_1, \ldots hecho_n$

Todo hecho exportado ha de ser deducible dentro del módulo o bien importado; de no ser así el módulo será considerado erroneo.
Distinguimos las declaraciones de los hechos que son importados, obtenidos en tiempo de ejecución, de los exportados, debido a su naturaleza distinta desde el punto de vista de la abstracción funcional. Al declarar un hecho como exportado indicamos que será obtenido en el módulo que lo declara como exportado, sin entrar en detalles de cómo será obtenido. Al declarar un hecho como importado indicamos que no existe ningun módulo que, mediante una abstracción funcional, lo exporte y que el módulo obtendra este valor de la base de hechos que exista en tiempo de ejecución quien "exporta" este hecho es, en definitiva, el usuario, en tiempo de ejecución.

Reglas *lreglas*

Conjunto de reglas del módulo. Estas reglas deducen los objetivos marcados. Aquí los hechos han de tener en cuenta el camino de acceso a traves de los submódulos.

Metareglas *lmre*

Conjunto de metareglas que definen las estrategias de ejecución de los submódulos involucrados dentro de cada módulo mediante las operaciones que definiremos más adelante.

2.1.2 Declaraciones de estructuras

El lenguaje de modularización permite definir tres tipos diferentes de estructuras: *conjuntos*, ***especificaciones*** y ***módulos***. Estas estructuras son objeto de estudio detallado en los siguientes apartados.

2.2 Conjuntos

Ya que es necesario declarar las interfaces de los módulos, y estas son conjuntos de hechos que se importan y/o exportan, parece útil definir conjuntos de hechos con tal de facilitar estas declaraciones, que se han de entender pura y simplemente como definiciones de macros de lectura. Cuando el compilador encuentre, después de la declaración de un conjunto, una referencia a su nombre, el efecto será la sustitución del nombre por los elementos del conjunto. Veanse las definiciones de los siguientes conjuntos en Pneumon-IA:

```
Conjunto Analíticas-generales = (exudado-pleural: Booleano)

Conjunto Tinciones = (gérmen-esputo:[CGPP, CGPC, BGN,CBGN], gram  Booleano;
                      número-PMM-esputo-x100a: Numérico, número celepi-esputo x100a: Numérico; germen-
                      pleural: [CGPP, CGPC, BGN,CBGN])

Conjunto Cultivos = (germen-cultivo-esputo: [Neumococo, Klepsiella, Estafilococo,
                      Hemofilus]; crecimiento-cultivo-esputo: Booleano, germen-cultivo pleural  [Neumococo,
                      Klepsiella, Estafilgcoco, Hemofilus]; germen-cultivo sangre  [Neumococo, Klepsiella
                      Estafilococo, Hemofilus]; cultivo-pleural + : Booleano, cultivo sangre +  Booleano)
```

2.3 Especificaciones

Una especificación es una descripción de las interfaces de los módulos en función de qué pueden ver del resto del programa y de qué aportan al resto del programa. Las especificaciones definen las relaciones entre los módulos. Es obligatorio definir las especificaciones antes de definir módulos que sigan el patrón por ellas definido. De esta manera podremos comprobar la corrección de los módulos definidos. A partir de la definición de un módulo se puede inferir cual es la especificación asociada. De hecho la labor que el compilador ha de realizar es comprobar que la especificación así inferida corresponda a la exigida en la declaración del módulo. Veanse los siguientes ejemplos:

```
Especificación Pneum =
Inicio
        Exporta  = (neumococo:Difuso)
Final

Especificación C-C-R =
Inicio
        Módulo CL:pneum
        Módulo RX: Radiol
        Exporta:(neumococo:Difuso)
Final

Especificación Radiol =
Inicio
        Módulo Radiol-favor:
                Inicio
                        Importa = Radiología
                        Exporta = (neumococo-favor:Difuso)
                Final
        Módulo Radiol-contra
                Inicio
                        Importa = Radiología
                        Exporta = (neumococo-contra.Difuso)
                Final
        Exporta = (neumococo:Difuso)
Final
```

Las especificaciones reflejan la estructura de submodulos que tienen los módulos; por ejemplo la segunda especificación declara que los módulos que la satisfagan tendrán que tener un submódulo que satisfaga la especificación *Pneum* y otro que satisfaga la especificación *Radiol*.
La declaración de un submódulo en la especificación de un módulo implica que este submódulo será visible desde fuera del módulo. Es decir, cualquier módulo que incorpore un módulo que siga la especificación C-C-R verá un submódulo de él llamado CL de especificación *Pneum*.
Si un módulo definido dentro de otro quiere ser escondido al exterior, la forma de hacerlo es no declarandolo en la especificación del módulo que le engloba. En el siguiente ejemplo se ve como la especificación *Exploración* no define ningun submodulo, en cambio en el módulo *Exp-físicas*, que cumple esta especificación, aparece el submódulo *BA = Bacterianicidad*. Esto implica que cualquier módulo que utilizara el módulo *Exp-físicas* no tendria derecho de acceso al módulo BA.

```
Especificación Exploración =
Inicio
        Importa = Sintomatología
        Exporta = (neumococo , Difuso)
Final
Modulo Exp_fisicas.Exploración =
Inicio
        Modulo BA = Bacterianicidad
        Reglas
        R1 Si herpes-labial y Ba   > Bacteriana  -+indica camino de acceso
            Entonces Neumococo es muy-posible
        R2 Si monoartritis-infecciosa y BA  -> Bacteriana > posible
            Entonces Neumococo es posible
        R3 Si esputo = herrumbroso y BA -> Bacteriana > moderadamente posible
            Entonces Neumococo es muy-posible
        R4 Si periodos-escalofrios = 1 y fiebre > 38 5 y BA-> Bacteriana
            Entonces Neumococo es posible
        R5 Si artritis y cardinal (articulaciones-afectadas) = 1
            Entonces monoartritis-infecciosaes ligeramente-posible
Final
```

Se puede ver facilmente que a partir de la definición de un módulo se puede extraer su especificación más general (exportacion de todos los submódulos y de todos los hechos deducidos e importación de todos los hechos no deducidos en él). El sistema de comprobación de la corrección de la definición hace un careo entre la especificación inferida y la especificación exigida para ese módulo. La especificación inferida tendrá que *respetar* a la especificación exigida como indica la siguiente definicion:

Def una especificación S_1 *respeta* una especificación S_2 si
1) Los hechos importados por S_1 son un subconjunto de los hechos importados para S_2
2) Los hechos exportados por S_1 son un superconjunto de los hechos exportados por S_2
3) Los submódulos exportados por S_1 son un superconjunto de los submódulos exportados por S_2

2.4 Módulos

La definición de módulos, tal y como se puede ver en la sintaxis, tiene la siguiente forma

Módulo *idmod*[(*listaparam*)] [: *exprespec*] = *expremod*

donde *expremod* puede ser una de las siguientes expresiones.

a) Un conjunto de declaraciones encapsuladas con un alcance limitado
b) Un identificador de módulo previamente definido
c) Una aplicacion de un módulo generico

2 4 1 Declaraciones encapsuladas

El siguiente módulo es un ejemplo de un conjunto de declaraciones encapsuladas:

```
Modulo Exp fisicas.Exploración =
Inicio
        Módulo BA = Bacterianicidad
        Reglas
        R1 Si herpes labial y Ba  > Bacteriana
            Entonces Neumococo es muy-posible
        R2 Si monoartritis-infecciosa y BA - > Bacteriana > posible
            Entonces Neumococo es posible
        R3 Si esputo = herrumbroso y BA  > Bacteriana > moderadamente-posible
            Entonces Neumococo es muy-posible
        R4 Si periodos-escalofrios = 1 y fiebre > 38 5 y BA- > Bacteriana
            Entonces Neumococo es posible
        R5 Si artritis y cardinal (articulaciones-afectadas) = 1
            Entonces monoartritis infecciosas es ligeramente-posible
Final
```

Dentro de las declaraciones aparece un submódulo que es igual a un módulo definido previamente de nombre *Bacterianicidad* y un conjunto de reglas. No es necesario declarar los hechos importados y exportados, puesto que estos ya lo fueron en el momento de definir la especificación *Exploración*, (ver ejemplo anterior). Si se quisieran escribir se habría de verificar la relación *respetar*, definida en el apartado anterior, con la especificación *Exploración*.

2.4.2 Declaraciones con identificadores de módulo

La declaración de submódulo en el módulo Exp-fisicas del apartado anterior es un ejemplo de este tipo de declaraciones. Es la forma de hacer accesibles dentro de un módulo todos los hechos y submódulos definidos dentro de otro. Así, los hechos definidos en el módulo *Bacterianicidad* son accesibles dentro del módulo *Exp-fisicas*. La forma de acceder a los hechos se hace mediante un prefijo que indica el camino de acceso al hecho. La declaración:

Módulo BA = Bacterianicidad

además de declarar que se quiere tener acceso a todos los hechos definidos en el módulo Bact-atip se proporciona el nombre que servirá para prefijarlos: BA. Si quisieramos utilizar el mismo nombre que cuando el módulo fue declarado, escribiríamos:

Módulo Bacterianicidad = Bacterianicidad

o bien su equivalente

Heredar Bacterianicidad

Esta prefijación sirve para distinguir distintas instancias del mismo hecho en módulos distintos. Así, por ejemplo la siguiente definición de módulo:

```
Modulo Combinación clínica-C C R =
Inicio
        Modulo CL = Clínica
        Modulo RX = Radiología
        Reglas
            R1 Si CL  > neumococo Entonces neumococo es seguro
            R2 Si RX  > neumococo Entonces neumococo es seguro
Final
```

CL y RX hacen referencia a dos módulos definidos anteriormente y que exportan el hecho ***Neumococo*** Como se puede ver la distinción entre el hecho *neumococo* de los dos módulos en las reglas R1 y R2 es posible gracias a la distinta prefijación de ambos hechos En este caso los dos hechos ***neumococo*** hacen referencia a la evidencia obtenida a partir de datos de tipo *clínico* y la evidencia a partir de datos de tipo *radiológico*. Este módulo combina las evidencias obtenidas de los dos submodulos
La prefijación se hace escribiendo el nombre del submódulo que contiene el hecho seguido de '->' y el nombre del hecho Por ejemplo:

```
Modulo A =
Inicio
        Modulo B =
        Inicio
                Modulo C =
                Inicio
                        Exporta hecho
                    {hecho se accede aqui como: hecho}
                Final
            {hecho se accede aqui como: C->hecho}
        Final
      {hecho se accede aqui como: B->C->hecho}
Final
```

Si se quiere tener acceso a todos los hechos definidos dentro del módulo, exportados y no exportados y sin tener que prefijar sus nombres se puede utilizar la primitiva:

Abrir *idmódulo*

esta primitiva se puede interpretar como una macro de lectura que efectua una copia de la declaración del módulo *idmódulo* dentro del cuerpo de definición del módulo donde se encuentra.

2.5 Módulos genéricos

Un módulo genérico es una abstracción de módulos que poseen una estructura parecida [Futatsugi et al, 1987]. Por ejemplo veanse los dos módulos siguientes:

```
Modulo Antecedentes-patológicos =
Inicio
        Modulo TM = Terreno-enfermo (Bacterianicidad)
        Modulo FM = Fármacos (Bacterianicidad)
        Exporta: neumococo;
        Reglas
                R1 Si no-dem (TM->neumococo) y FM->neumococo
                        Entonces neumococo es seguro
                R2 Si TM->neumococo y no-dem (FM->neumococo)
                        Entonces neumococo es seguro
```

```
                R3 Si TM -> neumococo y FM -> neumococo
                        Entonces neumococo es seguro
Final

Modulo Clinica =
Inicio
        Modulo AN = Antecedentes patologicos
        Modulo EX = Exp fisicas
        Exporta neumococo;
        Reglas
                R1 Si no-dem (AN -> neumococo) y EX -> neumococo
                        Entonces neumococo es seguro
                R2 Si AN -> neumococo y no-dem (EX -> neumococo)
                        Entonces neumococo es seguro
                R3 Si AN -> neumococo y EX -> neumococo
                        Entonces neumococo es seguro
Final
```

La estructura de los módulos es idéntica excepto los submódulos que cada una posee. Lo que realizan es una combinación de las evidencias aportadas teniendo en cuenta los casos en que en algunos módulos no se ha podido deducir nada. En el caso que ninguno de los modulos produjera ninguna evidencia sobre el hecho *neumococo*, no se aplicaría ninguna regla y el valor de neumococo exportado sería *desconocido*. Parece más correcto definir un modulo generico que se particularice en los submódulos concretos de cada caso. De esta manera escribiremos el codigo del módulo una sola vez y evitaremos inconsistencias. Así, el modulo generico abstraccion de estos dos sería el siguiente:

```
Módulo Combinación-y (X: Pneum, Y: Pneum) Pneum
Inicio
        Exporta: Neumococo,
        Reglas
                R1 Si no-dem X -> Neumococo y Y -> Neumococo
                        Entonces Neumococo es seguro
                R2 Si X -> Neumococo y no-dem Y -> Neumococo
                        Entonces Neumococo es seguro
                R3 Si X -> Neumococo y Y -> Neumococo
                        Entonces Neumococo es seguro
Final
```

Pneum es la especificación que engloba los módulos que exportan *neumococo*. Los módulos abstraidos pasarían a construirse de la siguiente manera:

```
Módulo Antecedentes-Patológicos = Combinación-y (Terreno enfermo (Bacterianicidad) Fármacos
(Bacterianicidad)

Módulo Clínica = Combinación-y (Antecedentes patológicos, Exp-físicas)
```

El módulo genérico se aplica sobre dos módulos que siguen la especificación *Pneum* y devuelve un módulo que también sigue la especificación *Pneum*. Esta descripción de los tipos de los argumentos y del resultado se hace con el fin de realizar la comprobación de los parametros actuales y de los resultados.
Los módulos genéricos tienen derecho a acceder únicamente a los hechos exportados por sus parámetros. Si se quisieran incorporar los submódulos de manera que fueran exportados se tendrían que declarar como submódulos de la siguiente manera:

```
Módulo Combinación y (X:Pneum; Y Pneum): Pneum1
Inicio
        Módulo Primero = X
        Módulo Segundo = Y
        Exporta Neumococo,
        Reglas
                {Conjunto de reglas}
Final
```

donde *Pneum*1 sería la especificación que extendería la de *Pneum* con dos submódulos de especificación *Pneum* y de nombres *Primero* y *Segundo*.

Hasta aquí hemos presentado las líneas básicas de la técnica de modularización utilizada en MILORD. Como se habrá podido comprobar ésta es independiente del tipo de reglas que se utilice y por lo tanto reutilizable sobre cualquier otro lenguaje basado en reglas; para una descripción detallada de MILORD ver [Sierra, 89].

3. ESTRATEGIAS

El concepto de estrategia tiene multiples visiones según el área de aplicación. En lo que aquí concierne entenderemos por estrategia la modificación en tiempo de ejecución de un módulo con el objetivo de aumentar la eficiencia y la adecuacidad de ejecución
Las modificaciones actuarán únicamente sobre el conjunto de submodulos de un módulo dado, modificando su estructura o bien eliminando completamente un submódulo considerado no necesario.
La semántica de este tipo de operaciones cabe buscarla en la adaptación de los programas al estado de conocimiento sobre un problema dado. Así, será lógico modificar un módulo que contenga un submódulo sobre problemas asociados a los transplantes cuando nos encontremos delante de un enfermo no transplantado. La programación inicial, de caracter necesariamente global, puede adaptarse así en tiempo de ejecución al estado de conocimiento cambiante.
La restricción imprescindindible de exigir, y que nos permite conservar las propiedades que las especificaciones de programas nos proporcionan, es la monotonía decreciente de las signaturas inferidas en los módulos modificados (ver fig. 3.1).
Las estrategias vienen representadas mediante metareglas de estrategia (mre) de la forma:

Si *condición* ENTONCES {expresión modular (EM)}

Donde:

condición representa un equivalente exacto a las premisas en las reglas de producción.

EM será la expresión que pasará a ejecución siempre que la condición sea satisfecha en el proceso de evaluación de las metareglas.

Sig (Mod) representa la signatura inferida de Mod. las estrategias Modifican los módulos respecto a los componentes principales.

* Modificación de submódulos
* Eliminación de reglas
* Eliminación de mre

En concreto toda mre puede ser aplicada una sola vez, y una vez aplicada no pasará al módulo resultante.
Además, el que la signatura inferida de un módulo modificado esté incluida en la signatura del original nos asegura que no se introduce nuevo código, es decir, que no creamos código en tiempo de ejecución. Debe respetarse, asimismo, que la signatura forzada sobre Mod (Fig 3.1) pueda serlo tambien sobre Mod', es decir: $Mod.\Sigma \Rightarrow Mod'.\Sigma$;en caso contrario habríamos violado la restricción impuesta sobre Mod. Tenemos que pensar que desde el punto de vista

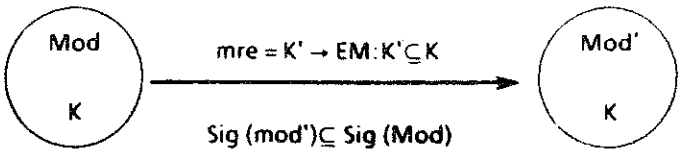


Figura 3.1: Acción de una estrategia sobre un módulo Mod.
Con un estado de conocimiento K

externo las modificaciones han de ser imperceptibles y por lo tanto no pueden afectar a la signatura impuesta sobre un módulo dado.
Esta aproximación al metacontrol, representado por las mre, posee caracteristicas reflexivas, a diferencia de la arquitectura de control multi-nivel no reflexiva utilizada en MILORD
A continuación haremos un repaso a los operadores autorizados en la construccion de expresiones modulares. Estos operadores son presentados como muestra del tipo de operaciones factibles con esta aproximación.

4.- OPERADORES DE GENERACIÓN DE ESTRATEGIAS

Tal y como hemos definido las estrategias en el apartado anterior, es necesario proporcionar operadores que permitan modificar los módulos y estudiar las propiedades de los módulos resultantes después de su aplicación. Especial interés representa el analisis de las signaturas del módulo original y del resultante.
Las operaciones que se proponen están relacionadas con operaciones conjuntistas sobre los elementos que comparan las signaturas. Así, una signatura está definida por cuatro componentes: un identificador, la lista de signaturas de los submódulos, con la jerarquia correspondiente, la lista de hechos importados, y la lista de hechos exportados; la representaremos por (N, M, I, E), donde N es el identificador de la signatura, los dos últimos conjuntos representan conjuntos de identificadores de hechos y donde M es el conjunto de signaturas de los submódulos. Así tendríamos por ejemplo:

(M1, ((M2, nil, externo, neumococo)), (externo), (neumococo))

La función *Arb*:M→Arboles nos devuelve la representación en forma arborea del conjunto de identificadores de M.
La definición de respetabilidad entre signaturas del apartado 2.3 , puede ser expresada ahora como:

Def. Diremos que $\Sigma_1 = (N_1, M_1, I_1, E_1)$ respeta $\Sigma_2 = (N_2, M_2, I_2, E_2)$ y notaremos $R(\Sigma_1,\Sigma_2)$ sii

1.- $E_2 \subseteq E_1$
2.- $I_2 \subseteq I_1$
3.- $\forall \Sigma' = (N_i, M_i, I_i, E_i) \in M_1$ y $\Sigma'' = (N_j, M_j, I_j, E_j) \in M_2$ tal que $N_i = N_j$ se cumple $R(\Sigma', \Sigma'')$

Esta respetabilidad es la propiedad que deseamos se mantenga entre la signatura inferida de los módulos después de aplicada una mre y la signatura que marcaba el modulo original

Def. Diremos que una operación Op sobre el universo de módulos M Op: M→M aplicada sobre un Módulo m de signatura Σ és correcta según respetabilidad sii $R(Sig(Op(m)), \Sigma)$

Las operaciones respetables definidas por nuestro sistema son las siguientes: Unión Intersección y diferencia

Sea M = (N, M, I, E) un módulo. Utilizaremos la misma representación de signaturas para los modulos, ya que para la definición de las propiedades que nos interesa destacar nos es suficiente, así pues se dará que Sig(m) = m.

UNION

Unión: $U_{ij}(M) = M' = (N, \{M_K\}_{K \neq (i,j)} + U(M_i, M_j), I, E)$, conservamos los módulos no unidos.

$U(M_i, M_j) = (N_i N_j, DIF(A,B) + UNI(A,B), I_i \cup I_j, E_i \cup E_j)$, si $N_i \neq N_j$
$= (N_i, DIF(A,B) + UNI(A,B), I_i \cup I_j, E_i \cup E_j)$, si $N_i = N_j$, donde
$M_i = (N_i, A, I_i, E_i)$
$M_j = (N_j, B, I_j, E_j)$
$DIF(A,B) = \{ (N_x,M_x,I_x,E_x) \in A / (N_x,P,Q,R) \notin B$ para algún P,Q,R$\} +$
$+ \{ (N_x,M_x,I_x,E_x) \in B / (N_x,P,Q,R) \notin A$ para algún P,Q,R$\}$
$UNI(A,B) = \{ U((N_x,M_x,I_x,E_x), (N_x,M_y,I_y,E_y)) / (N_x,M_x,I_x,E_x) \in A$ y $(N_x,M_y,I_y,E_y) \in B \}$

La unión de dos submódulos se interpreta como una operación de objetivos y código. Los elementos de exportación se unen, así como los de importación, los submódulos de ambos submodulos unidos pasan directamente al nuevo módulo, excepto aquellos submódulos que coincidan en el nombre, sobre los cuales se aplica recurrentemente la operación. El código (reglas) de los módulos se concatena para generar el código del módulo resultante, el identificador del nuevo módulo será la concatenación de los identificadores de ambos módulos; toda mre que haga referencia a los antiguos modificadores no será pasada al nuevo módulo, así como en el código resultante (reglas) se cambiarán las referencias a los antiguos identificadores de módulo para referenciar el nuevo. Para que esta operación sea correcta los módulos M_i y M_j no deben pertenecer a la signatura Σ de M. Es evidente comprobar que si $M:\Sigma$ entonces $M': \Sigma$

Por ejemplo, Sea el módulo M:

```
Modulo M: inicio modulo A final =
inicio
        Módulo A =
        Modulo B1 =
        inicio
            Módulo C =
            Módulo D =
                inicio
                    exporta = F1, F3
                    Reglas =
                final
            exporta = F2
            Reglas =
                    R1 Si D-> F1 entonces F2 es muy posible
                    R2 Si D-> F3 entonces F2 es posible
        final
        Módulo B2
        inicio
            Módulo D
                inicio
                    exporta = F3, F8
                    Reglas =
                final
            exporta = F4
            Reglas =
                    R1 Si D-> F3 entonces F4 es posible
```

R_i Si D->F_8 **entonces** F_4 es muy posible

final

importa = externo

exporta = F_5

Reglas

R_1 Si B_1 ->F_3 o B_2->F_4 **entonces** F_5 es muy posible

R_2 Si externo **entonces** F_5 es posible

final

expresaremos la operación sobre la sintaxis U_{B_1,B_2} (M) =

Módulo M: **inicio módulo** A **final** =

inicio

Módulo A =

Módulo B_1B_2 =

inicio

Módulo C =

Módulo D =

inicio

exporta = F_1, F_5, F_8

Reglas = {concatenacion de las reglas de ambos modulos de nombre D}

final

exporta = F_3, F_4

Reglas = {concatenación de las reglas de B_1 y B_2}

final

importa = externo

exporta = F_5

Reglas

R_1 Si B_1B_2->F_3 o B_1B_2 >F_4 **entonces** F_5 es muy posible

R_2 Si externo **entonces** F_5 es posible

final

Observese que la signatura inicio módulo A final ha sido respetada, y que la semántica global se ha mantenido. El único punto delicado de esta operación son los posibles efectos laterales que la concatenación de códigos puede provocar.

INTERSECCION

Intersección: $\cap_{ij}$ (M) = (N,{M_K}$_{K\neq i,j}$ + $\cap(M_i, M_j)$, I, E'), conservamos los módulos no intersecados

$\cap(M_i, M_j)$ = ($N_i N_j$, DIF(A,B) + UNI(A,B), $I_i \cup I_j$, $E_i \cap E_j$), si $N_i \neq N_j$

= (N_i, DIF(A,B) + UNI(A,B), $I_i \cup I_j$, $E_i \cap E_j$), si $N_i = N_j$, donde

M_i = (N_i, A, I_i, E_i)

M_j = (N_j, B, I_j, E_j)

DIF(A,B) = { (N_x,M_x,I_x,E_x) $\in$ A / (N_x,P,Q,R) $\notin$ B para algún P,Q,R} +

+ { (N_x,M_x,I_x,E_x) $\in$ B / (N_x,P,Q,R) $\notin$ A para algún P,Q,R}

UNI(A,B) = { U((N_x,M_x,I_x,E_x), (N_x,M_y,I_y,E_y)) / (N_x,M_x,I_x,E_x) $\in$ A y (N_x,M_y,I_y,E_y) $\in$ B }

E' = E - {M_i->e / e $\notin$ E_j} - {M_j->e / e $\notin$ E_i}

Esta operación es posible únicamente si:

- La parte de exportación Ex de la signatura Σ (M:Σ) cumple Ex$\subseteq$E'$\subseteq$E
- M_i y M_j $\notin$ Mod(Σ), módulos de Σ.

La intersección de los módulos es una operación de intersección de objetivos. Los elementos de exportación se intersecan. Los de importación en cambio permanecen todos, asi como los módulos se unen como en la operación de unión. La diferencia notable está en el aspecto del código, del cual desaparecen las reglas que concluyen hechos no exportados por el módulo

resultante. El resto de consideraciones son iguales que en la operacion de unión
Vease el resultado de la operación:

$\cap_{B_1,B_2}$ (M) =

Módulo M **inicio módulo** A **final** =
inicio
Módulo A =
importa = externo
exporta = F_3
Reglas = R_1 Si externo **entonces** F_3 **es** posible
final

La intersección de los módulos B_1 y B_2 da un conjunto de elementos de exportación vacio, por lo que el módulo no es considerado; al mismo tiempo la regla referente a B1 y B2 desaparece al hacer referencia a hechos exportados por un módulo inexistente.
Si en algún caso se violara que M': Σ la operación sería considerada erronea y así seria notificado.

DIFERENCIA

Diferencia: $/_{ij}$ (M) = = (N,$\{M_K\}_{k\in I-\{i,j\}}$ + /(M_i, M_j), I, E'), conservamos los módulos no restados

/(M_i, M_j) = (N_i, DIF(A,B) + UNI(A,B), $I_i \cup I_j$, E_i / E_j), donde
M_i = (N_i, A, I_i, E_i)
M_j = (N_j, B, I_j, E_j)
DIF(A,B) = { (N_x,M_x,I_x,E_x) $\in$ A / (N_x,P,Q,R) $\notin$ B para algún P,Q,R} +
+ { (N_x,M_x,I_x,E_x) $\in$ B / (N_x P,Q,R) $\notin$ A para algún P,Q,R}
UNI(A,B) = { U((N_x,M_x,I_x,E_x), (N_x,M_y,I_y,E_y)) / (N_x,M_x,I_x,E_x) $\in$ A y (N_x,M_y,I_y,E_y) $\in$ B }
E' = E-{M_i->e:e $\in$ E_j}

Esta operación es posible únicamente si la parte de exportación Ex de Σ cumple Ex $\subseteq$ E' $\subseteq$ E, y si cumpliendose que si $M_i \in$ Mod(Σ) entonces M' : Σ.

El resto de consideraciones son similares a las operaciones anteriores, excepto que en este caso se conservan los identificadores de todos los módulos.

5.- CONCLUSIONES

Hemos presentado una técnica de modularización aplicable sobre cualquier lenguaje basado en reglas. Esta Modularización se basa en la relación entre especificaciones (signaturas) y Módulos.La utilización de estrategias de resolución de problemas, fundamental en muchas de las aplicaciones en Inteligencia Artificial puede tener, en nuestra opinión, una solución muy interesante dentro de la propia metodología de modularización. Así, hemos definido las estrategias como modificaciones de los módulos que respeten su signatura Diferentes operadores de combinacion y modificación de módulos han sido presentados sin intención de agotar todas las posibilidades

La utilización de elementos procedentes de las técnicas de desarrollo de software más avanzado pueden ser de una enorme utilidad en la solución de los cuellos de botella que la IA tiene planteados hoy en día. Asimismo, la problemática característica de los sistemas de IA, sobre todo en los aspectos referentes al control, puede enriquecer estas técnicas para alcanzar un punto de encuentro muy fructifero entre ambos campos.

BIBLIOGRAFIA

Futatsugi K., Goguen J., Messeguer J., ada K. (1987): "Parametrized Programming in OBJ2", a AAVV: Proceedings of Ninth Int. Conference on Software Engineering, IEEE Comp. Soc. Press, pp 51-60

Harper R., McQueen D., Milner R. (1986): "Standard ML" Report, ECS-LFCS-86-2, Edinburg University.

Miller D.A. (1986) "A Theory of Modules for Logic Programming", a AAVV proceedings of 1986 IEEE Symp on Logic Programming.

O'Keefe R., (1985): "Towards an Algebra for Constructing Logic Programs", a AAVV: Proceedings of 1985 IEEE Symp. on Logic Programming, pp 152-160.

Shriver B., Wegner P. (Eds.)(1987): Research directions in Object-Oriented Programming, MIT Press.

Sierra C.A., (1989): Milord: Arquitectura Multi-Nivell per a Sistemes Experts en Classificació.Tesis Doctoral en Inteligencia Artificial por la Universitat Politècnica deCatalunya.

Sticklen J., Smith J.W. Chandrasekaran B., Josephson J.R. (1987): "Modularity of Domain Knowledge", International Journal of Expert Systems, Vol 1,1, pp.1-15.

Verdaguer A. (1989): PNEUMON-IA :Desenvolupament d'un sistema expert d'ajuda al diagnòstic mèdic, Tesis doctoral en medicina por la Universitat Autonoma de Barcelona.