DOCUMAT-90

Una metodología y un lenguaje para la ingeniería del conocimiento.

J. Agustí-Cullell and C.Sierra

CEAB
Camí de Santa Bàrbara
17300 Blanes, Girona, Spain.
e-mail: Sierra@ceab.es Agusti@ceab.es

RESUMEN

En esta comunicación se considera que el objetivo final de la Ingenieria del Conocimiento (IC) es asegurar la especificación adecuada de un problema (una Base de Conocimientos (BC) adecuada), al contrario que en la ingeniería software, donde la especificación formal es el punto de partida. Así, la metodología de IC debe soportar el peso de la creación y manipulación de un espacio en evolución de BCs cada vez más adecuadas. La metodología que aquí proponemos parte de tres componentes básicos: una noción precisa de BC parcial o módulo, un conjunto de operaciones de composición de módulos y la modificación de un módulo mediante refinamientos. El proceso de diseño y desarrollo de una BC se representa mediante un Grafo Dirigido Acíclico (GDA) cuyos arcos simbolizan tanto las operaciones de composición como las de refinamiento, y cuyos nodos simbolizan los módulos.La creación y manipulación de este grafo determina el lenguaje COLAPSES y el entorno de programación. . Las unidades básicas de COLAPSES son módulos y módulos genéricos. El lenguaje proporciona tres unidades básicas de manipulación de los módulos 1) refinamiento , incluyendo satisfacción, refinamiento herencia y ocultación de información, 2) composición a través de las declaraciones de submódulos y 3) aplicaciones de módulos genéricos a otros módulos. También se incluyen algunas nociones de la semántica del lenguaje y un modelo de ejecución.

Palabras clave: Desarrollo de bases de conocimientos; Reutilización de Software;Programación modular; Refinamiento de bases de conocimientos.

Investigación parcialmente financiada por el proyecto SPES CICYT 880j382 PICS "Diseño e implementación de un entorno para la especificación y construcción de S. E."

llamado **nivel de conocimiento** [Steels, 1989]. Aquí consideraremos los mismos conceptos pero utilizando terminología de programación[Goguen, 1986], en lugar de la terminologia de resolución de problemas .[Chandrasekaran, 1986], [Steels, 1989].

Programación Incremental

El método usual de la IC es la construcción de prototipos que serán extendidos o modificados mediante un proceso de evaluación de su comportamiento. Esta metodología ha de dar soporte a la especificación incremental, y ha de dar una definición de construcción incremental de prototipos. Esta definición ha de precisar la noción de interacción entre BCs y la noción de extensión de una BC. Estas definiciones evitarán un problema común en desarrollos de BCs: la imposibilidad de determinar las consecuencias de cualquier extensión de una BC.

El control de las interacciones en el proceso de construcción disminuirá drasticamente el coste de desarrollo de BCs y permitirá diseñar BCs de mayor complejidad.

Control

La programación del control se ha venido realizando comunmente desde un punto de vista global y en IC esta programación se realiza utilizando mecanismos de reflexión.[Maes, Nardi, 1988], [Akkermans et al, 1989], [Aiello, Levi, 1984]. La construcción incremental de BCs también requiere una programación incremental del control; esto significa que es necesario definir el control localmente, dentro de cada BC parcial. Así se asegura una estructuración del mecanismo de reflexión. De este modo el conocimiento de control es más específico y puede adaptarse a cada subproblema o BC parcial.I

Validación. Evaluación del comportamiento y verificación logica de BCs

El problema de la validación de Bcs ha sido tratado en otras comunicaciones [López et al, 1989]. Las técnicas de validación consideran principalmente dos problemas.1) la evaluación o análisis estádistico del comportamiento del sistema, comparandolo con su uso real o con otros expertos (idoneidad final) 2) la verificación sintáctica de la Bc para detectar y resolver errores estructurales y lógicos (consistencia). Estas dos propiedades sólo se han aplicado a BCs consideradas globalmente, sin tener en cuenta su proceso de construcción. Sin embargo creemos necesario considerar estas características integradas en el mismo proceso de construcción, por ejemplo, estudiarlas durante el proceso de construcción y refinamiento de las diferentes y sucesivas BCs parciales. La validación no debería ser un control de calidad final, sino que debería estar integrada en el proceso de construcción del sistema.

En esta comunicación explicaremos cómo todas estas características, deseables en la programación de BCs, se pueden lograr con la combinación de tres conceptos básicos: qué es exactamente una BC parcial, cuales son las operaciones de composición de BCs parciales y cuales son las modificaciones por refinamiento de una BC parciales. No nos extenderemos aquí sobre los aspectos de control, ya que esto nos llevaría a una comunicación demasiado larga. Sin embargo, indicaremos de paso las ventajas de la definición local del control.

2. Metodología propuesta

Toda nuestra metodología gira alrededor de la noción de módulo. Esta noción es una formalización de la idea informal de BC parcial. La unidad básica de desarrollo de una BC grande es el módulo (teoría) en contraposición con otras metodologías en las cuales la unidad básica es la regla (fórmula)[Fiadero, 1988], [Akkermans et al, 1989].Los módulos son las unidades del programa y consisten en: a) una interficie de entrada cuya función es dar una idea de cómo un módulo depende del exterior, b)una interficie de salida, que nos dice lo que el módulo aporta al exterior, y c) un cuerpo o núcleo, que contiene la definición del comportamiento del módulo. Este cuerpo puede estar inicialmente vacio, en cuyo caso, el módulo se convertiría en una pura interficie. Definir incrementalmente el cuerpo de un módulo es el punto central de la metodología de la programación incremental que proponemos.

Definimos dos tipos de operaciones sobre los módulos: Operaciones de combinación de diferentes módulos y una operación de refinamiento de un módulo. El proceso de desarrollo de una BC quedaría así representado con un Grafo Dirigido Acíclico (GDA) cuyos arcos

simbolizan tanto las operaciones de composición como la operación de refinamiento y cuyos nodos representan a los módulos (vease fig. 1) (el contenido de los módulos está explicado en el párrafo dedicado a la descripción del lenguaje COLAPSES). Las raices de este GDA representan las descripciónes más generales de la BC. Los nodos terminales representan las diferentes BCs parciales, concretas y ejecutables.

La manipulación de este GDA determina el entorno de desarrollo del lenguaje COLAPSES, en particular el compilador de este lenguaje verificará propiedades de las operaciones de construcción del GDA.

2.1. Operaciones de composicion

La actividad fundamental en el desarrollo de una BC grande es la estructuración de Bcs parciales (módulos) mediante operaciones de composición. El diseño de un lenguaje de composición de módulos es, por tanto, el objetivo clave de un lenguaje de programación de BCs. En esta comunicación presentamos una primera aproximación a un lenguaje de este tipo.

Las operaciones elementales de composición han de permitir tanto la construcción de un módulo mediante la unión de módulos previamente existentes cuyos componentes quedan protegidos, como la descomposición de un módulo en una serie de submódulos componenetes. Ambos métodos utilizan el mismo concepto de submódulo, que será presentado en uno de los párrafos de descripción del lenguaje. Por ejemplo, en la fig. 1, el módulo . *ext_date* está definido a partir de un grupo de submódulods todavía no definidos (en cursiva). (*type1_data...typen_data*) incluyendo cada uno de ellos un tipo de datos. En general las segmentos rectos muestran la dependencia de un módulo respecto a sus submódulos.

Para permitir una mayor reutilización de los módulos es necesario permitir al usuario la definición de sus propias operaciones de composición. Estas se realizan mediante el concepto de módulo genérico o paramétrico, cuyos parámetros especifican los módulos sobre los cuales este es aplicable.

Los módulos genéricos son por tanto funciones que se aplican sobre módulos dando lugar a nuevos módulos. También se pueden entender como una forma de descomposición de una problema en una parte abstracta, los parámetros del genérico, y una parte concreta, el cuerpo del genérico

Estas operaciones, unidas al concepto de módulo permiten llevar a cabo los requerimientos de una buena metodología de programación:

- La verificación se realiza modularmente, y, por tanto, su complejidad decrece drásticamente También es invariable respecto a las operaciones de composición.

- Los módulos son las unidades cuya reutilización es siempre posible mientras se se mantengan sus interficis Los módulos genericos son operaciones de combinación que permiten un mayor grado de reutilización.

- El carácter local del control, resulta del hecho de ser definido dentro de cada módulo; las operaciones de composición (submódulos) no afectan la localización del control. Las operaciones de composición aparecen indicadas mediante líneas delgadas en la fig. 1.

2.2. Operación de refinamiento

Esta operación realiza el concepto de programación incremental explicado anteriormente. A partir de un módulo permite la definición de un nuevo módulo cuyo contenido informativo es mayor conservando la interficie de salida del primero, es decir, conservando su comportamiento.

La operación de refinamiento se compone de las siguientes operaciones elementales: safisfactibilidad de interficies, herencia de componentes y ocultación de interficies..

La satisfactibilidad de las interficies garantiza que los módulos refinados van a tener el mismo comportamiento que el módulo del cual proceden por refinamiento.

La herencia ahorra la repetición de los componentes de un módulo ya definidos en el módulo anterior por refinamiento.

La ocultación de interficies se entiende como la ocultación de algunos de los componentes de un módulo, de manera que el módulo refinado mantiene la misma interficie de salida que el módulo del que procede. La ocultación de interficies permite la simplificación de las interacciones de los módulos y su limitación a aquella parte de comportamiento que se necesita en cada momento. En el grafo de la fig.1 esta operación aparece en trazo grueso.

2.3. Grafo de la BC

El grafo de la BC representa la historia global de desarrollo de la BC. Los nodos cercanos a la raiz del grafo muestran las decisiones de diseño más fundamentales y son temporalmente las más antiguas. Los nodos cercanos a las hojas del grafo son los que contienen las decisiones de implementación más concretas.

El ciclo de vida de una BC implica la aplicación de algunas operaciones de modificación del grafo. Algunos ejemplos elementales de esto son los siguientes: cambios de implementación, cambios de interficie y ocultaciones, refinamientos mediante módulos genéricos, etc. Algunas de estas modificaciones han sido ejemplificadas en la fig. 1. La definición de operaciones de modificación del grafo a partir de otras operaciones más elementales ya mencionadas es un punto central de nuestra investigación metodológica.

En la siguiente sección presentamos un lenguaje llamado COLAPSES diseñado especificamente para dar soporte a esta metodología de programación. Su semántica se ha formalidado utilizando técnicas de los lenguajes de especificación formal.; sin embargo, la descripción de dicha semántica no es el objetivo de esta comunicación. [Sannella, Wallen 1987].

Las operaciones básicas de construcción y modificación son independientes del lenguaje básico utilizado para definir los cuerpos de los módulos. Esta independencia permite la utilización de diferentes lenguajes de representación en los diferentes módulos.[Harper, 1989].Una aplicación sencilla de esta posibilidad es la utilización de diferentes lógicas multivaluadas en los diferentes módulos. [Agustí et al, 1990]. Aunque esta posibilidad es metodológicamente importante no la desarrollaremos aquí, ya que sería necesario dedicarle una comunicación completa

En la implementación actual de COLAPSES y en los ejemplos propuestos, el lenguaje básico utilizado es el lenguaje MILORD basado en reglas [Sierra, 1989]. En un futuro este lenguaje se ampliará con lógica de primer orden, tipos ordenados y un componente funcional.

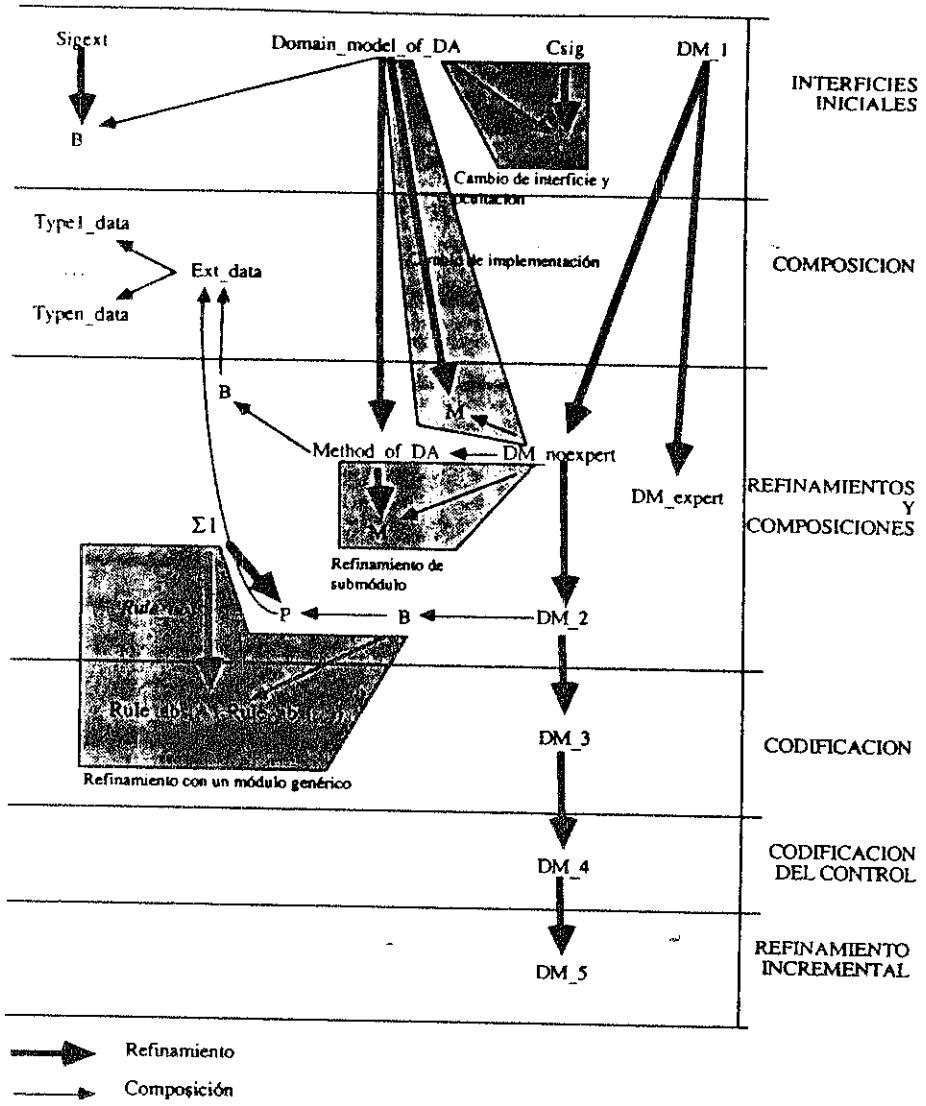

Figura 1. Ejemplo de representación del grafo (los contenidos de los nodos se muestran en las figuras 2, 3, 4, 5).

3. Lenguaje propuesto: COLAPSES

En esta sección se presentan las principales componentes de COLAPSES mediante un ejemplo. El ejemplo no tiene relación con ninguna aplicación; ha sido diseñado exclusivamente para introducir la sintaxis y la semántica del lenguaje. El ejemplo se introduce progresivamente, pero su estructura global aparece en la fig.1 . Aquellos elementos que no han sido completamente desarrollados, ya sea debido a la falta de espacio, o para mantener mayor generalidad, aparecen escritos en cursiva.

Las unidades básicas de la BC escrita en COLAPSES son los módulos. Estos pueden ser organizados jerarquicamente y consisten en un conjunto encapsulado de declaraciones de importación, de exportación, de reglas, de control, de meta-reglas y submódulos. Las primeras son las que dan contenido a los módulos y la declaración de submódulos define la estructura jerárquica de la BC. La declaración de submódulos es identica en todos sus aspectos a la declaración de módulos. (Vease fig.2).

El lenguaje tienen tres mecanismos básicos de manipulación de módulos:

1) Refinamientos de módulos, incluyendo la satisfacción, herencia y ocultación de información.
2) Composición de módulos mediante la declaración de submódulos y
3) Operadores de composición definidos por el usuario y llamados módulos genéricos

Las secciones 3.1 y 3.2 se referencian mutuamente.

3.1 Declaraciones primitivas

Los elementos más básicos de MILORD, el lenguaje básico de COLAPSES, son los hechos de orden 0+, las reglas de producción y las meta-reglas. La fig.2 muestra algunas de estas declaraciones primitivas.

IMPORTACION

Los hechos importados son aquellos cuyos valores se obtienen en tiempo de ejecución. Se distinguen de los hechos deducidos porque representan una situación especial, la de interacción con el usuario. Estos hechos son declarados por:

Import $fact_1; fact_2;...;fact_n$

y sus valores se obtienen mediante un predicado con semántica de entrada y salida llamado "Query". Con esta declaración definimos parte de la interficie de entrada del módulo que la contiene.

EXPORTACION

Los hechos exportados son aquellos que pueden ser usados por otros módulos. Definen parte de la interficie de salida de un módulo. Todos los hechos exportados deben ser conclusiones de reglas del módulo o bien ser importados desde otro módulo. Son declarados por:

Export $fact_1; fact_2;...;fact_n$

Las conclusiones de las reglas y los hechos importados no mencionados en la declaración de exportación quedan ocultados al resto de los módulos. Esta es la única declaración obligatoria en la construcción de un módulo. Un módulo al cual no se le pueden hacer preguntas, es decir, un módulo al que no se le pueden hacer preguntas, no tiene sentido.

NUCLEO

El núcleo se compone de dos partes llamadas conocimiento deductivo y conocimiento de control. El conocimiento deductivo incluye aquellas declaraciones del lenguaje básico, que en nuestra implementación son reglas de producción. El conocimiento de control está representado mediante un metalenguaje que actua por reflexión sobre el conocimiento deductivo y también sobre la jerarquía de módulos. La implementación actual del metalenguaje se limita al uso de meta-reglas y a la definición de algunos parámetros de control.(*backward*, *diving*, etc.). En un futuro, el metalkenguaje permitirá una programación estructurada e incremental del control. La programación incremental y la modificación de los módulos son las encargadas de dar gradualmente contenido al núcleo. Un módulo con un núcleo vacio puede ser considerado como una simple interficie.

```
Module Ext_data =
 Begin
        Module T1 = type1_data
        ...
        Module Tn = typen_data
        Import ext_data1; ext_data2; ...; ext_datam
        Export ext_data1; ext_data2; ...; ext_datam
        Deductive knowledge =
                R1 if Query(string1, values1, X1)
                  then ext_data1 = X1
                ...
                Rm if Query(stringm, valuesm, Xm)
                ...then ext_datam = Xm
        End deductive
        Control knowledge
                Deductive search is backward
                Structural search is diving
                Deductive control is ∅
                Structural control is
                MR1 if Query(string_there_is_type1, Boolean,
                false)
                  then eliminate(T1)
        End control
 End
```

Figura 2.

3.2 Declaración de Modulo

Las declaraciones de módulo tienen la siguiente forma:

Module *modidentifier* [:*modexpr1*][=*modexpr2*]

donde *modexpr2* puede ser:
a) un conjunto de declaraciones encapsuladas con visibilidad limitada.
b) un nombre de módulo definido en cualquier otro lugar o bien todavía sin definir.
c) la aplicación de un módulo genérico.
y *modexpr1* puede ser cualquiera de los elementos anteriores, excepto en el caso que defina un módulo desde el cual refinamos.

Donde el símbolo ":" representa la operación de refinamioento de módulos, y el símbolo"=" representa la operación de composición de módulos. Si no hay ninguna operación, entonces la declaración constituye sólo una definición de la interficie. En las próximas secciones explicamos detalladamente estas formas de declaración de módulo.

Declaraciones encapsuladas

El módulo *Ex_datu* (fig.2) es un ejemplo de un conjunto encapsulado de declaraciones. Contiene todas las declaraciones permitidas mencionadas anteriormente. Puede ser mostrado al usuario como un módulo estructurado jerarquicamente.

Declaraciones por referencia

Los nombres de los módulos se utilizan para nombrar otros módulos. Los módulos nombrados puede que todavía no hayan sido creados, ayudando así al diseño descendente.
Para representar la dependencia de un módulo A respecto a un módulo B escribimos:1

```
Module A
      Begin   Module X=B
      End
```

El módulo *Method_of_DA* (fig 3) referencia al módulo *Ext_data* y lo renombra *B* Esta declaración hace que todos los hechos exportados por*Ext_data* sean visibles en el núcleo de *Method_of_DA* . De esta manera se puede ver que las reglas de este último módulo usan en sus premisas hechos definidos por el otro módulo, prefijándolos con el identificador del módulo que los exporta. El símbolo de prefijado es "/". La concatenación de prefijos permite representar el camino hasta un hecho dentro de la jerarquía de módulos. El prefijado es útil para distinguir entre diferentes instancias de los hechos, que pueden tener así asociadas distintas definiciones.
Si no deseamos cambiar el nombre de un módulo referenciado como submódulo podemos utilizar la declaración Inherit (vease el modulo *DM_noexpert*). Para lograr que los hechos de un módulo sean directamente accesibles sin ningún prefijado debenos declarar al submódulo como abierto 'open' (vease el modulo *Rule_abs*). De hecho, las declaraciones open copian el núcleo y los submódulos del módulo abierto sin tener en cuanta su interficie. Esta es una operación que no proteje las interacciones dentro de un módulo, y debe ser usada con cuidadol
La semántica formal de la composición jerárquica de módulos se construye a partir de dos operaciones elementales. La unión de los cuerpos de los módulos y el renombramiento de los espacios de nombres protegiendo las interacciones entre los núcleos de los módulos.

```
Module Domain_model_of_DA =
 Begin
        Module B : Sigext
        Export DA1; ...; DAn
 End

Module Method_of_DA : Domain_model_of_DA =
 Begin
        Module B = Ext_data
        Deductive knowledge =
                R1 if B/ext_dataj and ... then DAi1
                ...
                Rm if B/ext_datak and ... then DAim
                ...
        End deductive
 End
```

Figura 3

Refinamiento de módulos

Refinamiento es una operación binaria entre módulos construida a partir de dos operaciones elementales: la herencia y la ocultación. y de un predicado de satisfacción de interficies. La sintaxis de esta operación ":" es:

Module *modid:modexpr* [= *modexpr'*]

El modulo *Method_of_DA* de la fig. 3 es un refinamiento del módulo *Domain_model_of_DA*. El segundo es una interficie del primero, es decir, indica qué hechos serán exportados y qué submódulos serán visibles desde el exterior. Un ejemplo de ocultación de un submódulo se muestra entre los módulos *DM_1* and *DM_noexpert* of de la fig. 4; De la misma forma los hechos deducidos y exportados pueden ser ocultados por el módulo refinado.

Cada identificador de módulo tiene asociada como valor una expresión de módulo (*modexpr*) en el entorno; La operación de refinamiento determina la expresión asociada *modid* a partir de las expresiones *modexpr* y *modexpr'* de la siguiente manera:

1.- Si modexpr'= ∅, entonces value(*modid*)=*modexpr* (usando la herencia).

2.- Si modexpr' ≠ ∅, entonces value(*modid*)=:(*modexpr*,*modexpr'*) (usando la herencia, la ocultación y la satisfacción).

La definición de la operación ":" es:

:(A,B)=**if** verify(A,B)
then
Inherit(Hide(C,B),B) where
C=A(s_i/:(s_i,s_j))
(S_i submodule of A;S_j submodule of B; modid(S_i)=modid(S_j))
else Module-error

* El predicado de satisfactibilidad"verify"permite la comprobación de muchas propiedades lógicas, tales como la como contradicción de las reglas, la no circularidad de las reglas por control de retroceso, etc. La implementación actual de este predicado :(A,B) comprueba:
* Que el módulo A no es visible si el B está oculto (considerar la recurrencia de módulos)
* Que los hechos importados por A son un subconjunto de los importados por B.

Notese que la operación está definida recurrentemente sobre la jerarquía de los módulos sobre los que opera. A cada nivel de la recursividad tienen lugar dos operaciones elementales:

- Ocultación. Hide:MxM----->M(donde M es el conjunto de módulos)
La interficie (hechos exportados, importados y los submódulos visibles desde el exterior), del segundo parámetro es impuesta sobre el primero.
El objetivo de esta operación es que todos los módulos refinados a partir de un módulo dado tengan un comportamiento equivalente.

- Herencia. Inherit : MxM----->M
El núcleo del módulo resultante es la unión de los núcleos. En caso de existir un conflicto en los parámetros de control, los parámetros del primer argumento prevalecen. Los submódulos del módulo resultante son los del primer argumento más aquellos del segundo cuyos identificadores no aparecen como identificadores de submódulo en el primero. Las interficies del módulo resultante son las del segundo argumento.

En el grafo de fig. 1 y en los módulos de la fig. 4 se muestran diferentes ejemplos de refinamiento. Los refinamientos *DM_noexpert:DM_1* y*DM_2:DM_noexpert* representan la programación incremental de la interficie de entrada. El refinamiento *DM_3:DM_2* representa un paso de codificación del componente deductivo. El refinamiento*DM_4:DM_3* es un paso de

codificación del componente de control. Finalmente el refinamiento *DM_5:DM_4* realiza un refinamiento incremental del componente deductivo.

Con estas operaciones de composición y refinamiento de módulos es posible modificar algunas propiedades de la estructura del grafo. Las propiedades más importantes son la transitividad del refinamiento y la distributibidad entre el refinamiento y las operaciones de composición. Estas propiedades nos seguran la computabilidad dentro del grafo a partir de su representación mínima.

```
Module DM_1 =
 Begin
       Export F;
 End

Module DM_Noexpert : DM_1 =
 Begin
       Inherit Ext_data
 End

Module DM_2 : DM_Noexpert =
 Begin
       Module B : P
 End

Module DM_3 : DM_2 =
 Begin
       Deductive knowledge
            R1 if B/H then F(a)
       End deductive
 End

Module DM_4 : DM_3 =
 Begin
       Control knowledge
            MR1 if Ext_data/ext_data_j
              then inhibit-rules-using(B/H)
       End control
 End

Module DM_5 : DM_4 =
 Begin
       Deductive knowledge =
            R2 if Ext_data/ext_data_k then F(b)
       End deductive
 End
```

Figura 4.

Aplicacion de módulos genéricos

La instanciación de un modulo genérico se considera como un declaración de módulo. Esto se explica en la siguientesección.

3.3 Declaración de módulos genéricos

La definición de módulos genéricos abre al usuario la posibilidad de definir operaciones de composición específicas. Esta técnica estandar consiste en aislar una parte de un programa, o módulo, de su contexto, y después hacer una abstracción especificando:

1) Aquellos módulos de los cuales depende el módulo abstraído (requerimientos o interficie de importación).
2) La contribución del módulo abstraído al resto del programa (resultado o interficie de exportación). La definición interna del módulo abstraído se hace en términos de las interficies de importación.

El ejemplo obvio de esta técnica es la programación funcional, donde tales abstracciones forman las unidades básicas del programa. El cuerpo de la función define cómo calcular los resultados en términos de las entradas o requerimientos. En la programación modular abstraemos conjuntos encapsulados de declaraciones del lenguaje básico (reglas). Estas abstracciones son, de hecho, funciones que actuan sobre programas y son llamadas módulos paramétricos o genéricos. (los tipos de los parámetros son las interficies de importación). Cuando aplicamos un módulo genérico a módulos concretos que satisfacen sus interficies de importación, el resultado es un nuevo módulo que satisface la interficie de exportación. El método natural para construir grandes BC consiste en aplicar módulos genéricos a módulos particulares construidos previamente.

La definición de módulos genéricos sigue el mismo esquema expresado en el ejemplo *Rule_abs* en fig.5. En este ejemplo se muestra un caso particular de abstracción.El módulo *P* contiene reglas que deducen el hecho *H* a partir de patrones similares en sus premisas. Resulta facil hacer una abstracción de este patrón haciendolo paramétrico sobre las diferencias. El resultado de esta abstracción es el módulo *rule_abs*, definido binariamente para hacerlo facilmente asociativo. Así, el módulo *p'* puede ser obtenido como la aplicación reiterada del módulo *rule_abs*. Este módulo genérico podrá ser reutilizado siempre que se necesite el mismo patrón de generación de módulos.

Para determinar mejor la semántica de los módulos genéricos precisaremos lo siguiente:

* Los parámetros de la definición se declaran como refinamientos entre nombres de variables (las cuales serán instanciadas por los módulos actuales) y expresiones modulares. Estas expresiones modulares garantizan una interficie de entrada mínima para poder definir el núcleo del módulo genérico. .

*Al aplicar un módulo genérico sobre módulos concretos se realiza la operación de refinamiento declarada en los parámetros. Entonces, y mediante técnicas de expansión de código se obtiene el módulo resultante.

* La aplicación de un módulo genérico sobre módulos concretos puede ser restringida por una declaración de compartir submódulos entre los parámetros actuales.Es decir, los módulos declarados como compartidos deben ser idénticos. [Agustí, Sierra, Sannella, 1989].

Queremos potenciar el proceso de construcción incremental de BCs mediante la aplicación de módulos genericos, de manera que, siempre que la definición de un módulo cambia, estos cambios queden reflejados en el resto del programa. La forma de llevar esto a cabo es simplemente repitiendo la aplicación de aquellos módulos genéricos que referencian al módulo modificado. El compilador puede automatizar este proceso de *relinking* de manera que el usuario queda libre de esta pesada tarea.


```
Module P : S1 =
 Begin
        Module Q = Ext_data
        Deductive knowledge =
                R1 if ... then A1
                ...
                Rn if ... then An
                Rn+1 if A1 and Q/B then H
                ...
                R2n if An and Q/B then H
        End deductive
 End

Module S1 =
 Begin
        Export H
 End

Module Rule_abs(X:Begin Export A End; M : S1) =
 Begin
        Open M
        Module Q = Ext_data
        Export H
        Deductive knowledge =
                R1 if X/A and Q/B then H
        End deductive
 End

Module P' : S1 =
        Rule_abs(A1_modul, Rule_abs(A2, ...
                                Rule_abs(An, Begin End)) ...)
```

Figura 5.

3.4 Gestión del entorno del grafo de desarrollo de la BC

El ciclo de desarrollo de la Bc consiste en la creación y manipulación del grafo mencionado anteriormente. La manipulación eficiente de este grafo debe ser realizada por un entorno de programación que incluya facilidades de visualización y de edición. Este entorno se está desarrollando y han sido identificadas algunas operaciones de manipulación del grafo . Entre los ejemplos de manipulaciones básicas (vease las partes sombreadas del grafo) tenemos las siguientes:

*Cambios de inteface y ocultación.
*Cambios de implementación .
*Refinamiento de la implementación de un submódulo
*Refinamiento mediante un módulo genérico

La identificación de nuevas operaciones de manipulación integradas en el entorno forma parte de las líneas de investigación futuras.

4. Semántica: modelo de compilación y ejecución .

La semántica del lenguaje ha sido definida en un estilo denotacional. Se han definido los objetos semánticos y principalmente son los siguientes:

- programa
- entorno
- código
- módulo

Las funciones semánticas más importantes son la generación y el linkaje, las cuales realizan la compilación de la sintaxis abstracta a los objetos semánticos.

La operación de generación crea el código asociado a un módilo concreto marcando su dependencia de otros módulos o bien por composición o bien por refinamiento. La operación de linkaje recibe pares de códigos asociados a los módulos e intenta resolver las interdependencias generando nueva estructura de código.Estas operaciones actuan como co-rutinas en la compilación de la jerarquía de módulos.

La construcción de la representación interna del grafo mediante los objetos semánticos tiene algunos aspectos que merecen una consideración especial. Hasta ahora los entornos de nombres del compilador eran construidos como pilas, añadiendo nuevos vínculos al descender en la estructura jerárquica y borrando vínculos al subir en la misma.[Agustí, Sierra, Sannella, 1989], [Hasselgren, 1989]. Esto no permitía la referencia a los componenetes que no hubieran sido definidos anteriormente (en un compilador de un paso como el nuestro). Nuestro compilador elimina esta restricción y permite la compilación separada.Además esto puede ayudar al editor en la construcción del grafo.

Los detalles del proceso de compilación pueden verse en el informe [Sierra, Agustí, 1990]. Veamos ahora algunos aspectos del modelo de ejecución de los módulos.

El modelo de ejecución de un módulo está basado en la ordenación de las reglas de producción y de los submódulos por un componente de metacontrol. Este componente de metacontrol permite también inhibir algunos submódulos y algunas reglas de un módulo.Esto se hace para adecuar un módulo a las necesidades concretas de ejecución. El metacontrol funciona mediante un mecanismo de reflexión implícita. [Maes, Nardi, 1988]. Se activa cuando se obtiene una nueva información proveniente tanto del exterior como de la parte deductiva del módulo o como un predicado exportado por un submódulo. La ejecución de un módulo consiste en la ejecución de su componente deductivo siguiendo el orden establecido por el metacontrol. Además del mecanismo de reflexión mencionado anteriormente la ejecución vienen determinada por los dos parámetros de control que indicamos a continuación:

*Búsqueda estructural: determina como se ejecutan los submódulos. Diving significa ejecución de los submódulos bajo demanda del componente deductivo es decir, cuando lo requiere la aplicación de un regla. Dragging significa que los submodulos son ejecutados antes de que se ejecute el componente deductivo.

*Búsqueda deductiva:determina el tipo de algoritmo utilizado en la deducción (Backward o Forward)

Es necesario notar que la ejecución diving implica la demostración de un predicado, mientras que la ejecución dragging implica la demostración de toda la interficie de exportación

Este modelo de ejecución sigue de cerca el modelo de MILORD, que es el lenguaje básico de COLAPSES.

5. Conclusión

Hemos analizado algunos requerimientos metodológicos de la ingeniería del conocimiento basandonos en nuestra experiencia en el desarrollo de BCs. Hemos mostrado que los conceptos de validación, reutilización, programación incremental y control reflexivo han de estar basados en una noción precisa de base de conocimientos parcial (el módulo). Hemos diseñado un lenguaje para la composición y refinamiento de módulos Los módulos genéricos van un poco más allá en el concepto de reutilización, permitiendo la construcción de grandes bases de conocimientos a partir de una biblioteca de módulos genéricos. El ciclo de desarrollo de una BC ha sido representado mediante un GDA, donde los nodos representan módulos y los arcos operaciones de composición y refinamiento. El ciclo de vida de este GDA implica la aplicación de algunas operaciones de modificación del grafo. La definición de operaciones potentes de modificación del grafo a partir de operaciones elementales constituye el núcleo de nuestra investigación metodológica futura. La manipulación eficiente de este grafo debería llevarse a cabo en un entorno de programación que incluyera facilidades de visualización y de edición..

Referencias bibliográficas

Agustí J., Sierra C., 1990 : "COLAPSES: A Methodology and a Language for Knowledge Engineering". CEAB Research Report 90/2.

Agustí J., Sierra C., Sannella D., 1989 : "Adding generic modules to flat rule-based languages: A low cost approach", in *Methodologies for Intelligent Systems, 4*, Elsevier, pp. 43-51.

Agustí-Cullell J., Esteva F., García P., Godo L., 1990 : "Formalizing Multiple-valued Logics as Institutions", CEAB Research Report 90/2.

Aiello L., Levi G., 1984 : The uses of metaknowledge in AI system, ECAI 84, pp. 705-717

Akkermans H., Wielinga B., Schreiber G., Balder J. 1989 : "Towards a Formal Specification of Knowledge Models", private communication.

Chandrasekaran B., 1986 : "Generic Tasks in Knowledge-Based Reasoning: High-level Building Blocks for Expert Systems Design", Research Report, Ohio State University.

Fiadero J., Sernadas A., Sernadas C., 1989 : Knowlledge Bases as structured theories, LNCS 338, pp. 469-485.

Goguen J. A., 1986 : "Reusing and Interconecting Software Components", *IEEE Computer*, February 1986, pp. 16-28.

Harper R., Sannella D., Tarlecki A., 1989 : "Structure and Representation in LCF", Proceedings of 4th IEEE Symp. on Logic of Computer Science.

Hasselgren R. A. 1989 : Modular construction of production rule-based systems, Art. Int. Dept. Report, University of Edinburh.

López B., Meseguer P., Plaza E., 1989 : An state of the art on KBS validation, *AI Communications*, to appear.

Maes P., Nardi D. 1988 : "Meta-level Architectures and Reflection", Elsevier Science Publishers B. V., Amsterdam.

Partridge D. 1986 : "Artifitial Inteligence. Aplication to Software Engineering", Ellis Horwood Ed., London.

Plaza E., López de Mántaras R., 1989: "Model-Based knowledge acquisition for heuristic classification systems", *SIGART Newsletter*, April 1989, N 108, pp. 98-105.

Sannella D., Tarlecki A., 1989 : Towards formal development of ML programs: methodological aspects and matematical foundations, LFCS Report, University of Edinburgh.

Sannella D., Wallen L. A., 1987 : A Calculus for the Construction of Modular Prolog Programs, Proceedings of 87 IEEE Symp. on Logic Programming, pp. 368-378.

Sierra C., 1989 : MILORD: Arquitectura meta-nivell per a sistemes experts en classificació, Ph. D. Thesis, Universitat Politècnica de Catalunya.

Steels L., 1989 : Components of Expertise, Art. Int. Lab. Research Report, Vrije Universiteit Brussel.

Van de Velde W., 1988 : "Learning from Experience", Ph. D. Dissertation of Vrije Universiteit Brussel.