Herramientas de construcción de sistemas expertos

Ingeniería del conocimiento

A. Rodríguez Rodríguez
J. Hernández Cabrera
A. Plácido Castro

Facultad de Informática
Universidad de Las Palmas de G.C.

Ingeniería del conocimiento

Herramientas de construcción de sistemas expertos
Volumen 3

Abraham Rodríguez Rodríguez
José Juan Hernández Cabrera
Ana María Plácido Castro

Contenidos

I Evaluación de herramientas de construcción de sistemas expertos

II Nexpert Object v.2.0

III ART-IM v.2.5

IV CBR Express v.1.1

CAPÍTULO I

Evaluación de Herramientas de Construcción de Sistemas Expertos

Índice

1 Introducción 1

2 Estructura de la evaluación 3
- 2.1 Panorámica 3
- 2.2 Características de la Aplicación 4
- 2.3 Capacidades de la Herramienta 8
- 2.4 Métricas 9
- 2.5 Técnicas de Evaluación 11
- 2.6 Contextos 13

3 Metodología 15
- 3.1 Los ocho pasos para la evaluación 16

4 Conclusiones 19

Bibliografía 21

1 Introducción

La proliferación en los últimos años de herramientas para la construcción de sistemas expertos ha añadido otro problema al desarrollo de dichos sistemas. Las herramientas son sistemas complejos y para llegar a conocerlas es necesario invertir gran cantidad de tiempo y dinero. De ahí la importancia de desarrollar un método que ayude a la evaluación y a la selección de la herramienta adecuada. La evaluación consiste en encontrar que características se adaptan mejor a una tarea determinada.

En este tema se muestra la estructura de un criterio de evaluación, y una metodología para la elección de una herramienta, dada una tarea a realizar. El método que se presenta también puede servir para descubrir la fortaleza o debilidad de una determinada herramienta que se quiera estudiar o desarrollar.

2 Estructura de la evaluación

La evaluación y la elección de una herramienta debe incluir todos los aspectos del dominio del problema, del problema en particular, y del proyecto a desarrollar. En esta sección nos centraremos en la estructura que se utilizará para la evaluación de las herramientas de sistemas expertos. La sección siguiente contiene la metodología que utiliza esta estructura.

Es posible aplicar la presente estructura sobre cualquiera de las herramientas disponibles en la actualidad y sobre las que se puedan ir desarrollando. No todas las capacidades que se proponen están disponibles en las herramientas existentes en este momento.

Aunque la estructura incluye una gran número de criterios, estos pueden simplificarse según sean las necesidades de un proyecto en particular. Además, la mayoría de las dimensiones pueden (o deben) priorizarse, de forma que la evaluación se pueda realizar más fácilmente.

2.1 Panorámica

La evaluación de las herramientas incluye cinco dimensiones importantes:

- *Características de la aplicación:* representan el contexto, su dominio y el proyecto.
- *Capacidades de la herramienta:* representa la funcionalidad de la herramienta que se está considerando.
- *Métricas:* son medidas que se utilizan para evaluar las capacidades de la herramienta.
- *Técnicas de evaluación:* son las formas de aplicar las métricas.
- *Contextos:* representan los requisitos necesarios para utilizar una herramienta en fases diferentes del desarrollo del sistema experto.

Estas dimensiones se utilizan de la siguiente manera para evaluar una herramienta para construir una aplicación dada:

> Dadas las características de la aplicación relevantes, aplicar las métricas mediante las técnicas de evaluación para evaluar una capacidad determinada de una herramienta en un contexto determinado.

O dicho de otra forma:

> Utilizar una técnica de evaluación
> para evaluar una métrica
> de una capacidad de una herramienta
> en un contexto determinado
> dadas las características de la aplicación.

Un concepto especialmente importante durante toda la evaluación es el de la integración. Puede ser visto como una característica de la aplicación (integración de la herramienta en el entorno de desarrollo), una capacidad de la herramienta (soportar la integración del sistema experto en un entorno existente), o una métrica a ser aplicada sobre otra capacidad de una herramienta (medir la calidad de la integración de los paradigmas representacionales).

2.2 Características de la Aplicación

Esta dimensión representa el impacto de la aplicación en la evaluación de la herramienta. Es importante determinar la cantidad mayor posible de las características siguientes para poder deducir las restricciones y requisitos apropiados.

Características de la Aplicación
- **a Características del Problema**
  - Dominio del Problema
    - Tipos de conocimiento
    - Restricciones
  - Problema a resolver dentro del Dominio
    - Procesamiento/conocimiento/representación especial
    - Tipo de problema
    - Otros atributos del problema
  - Adquisición del Conocimiento
    - Características y restricciones
  - Entorno Objeto
    - Restricciones
    - Usuarios finales

b Características del Proyecto
Ambito
Metas y presupuesto
Entorno de desarrollo
Restricciones
Equipo de desarrollo
Características

2.2.1 Características del Problema

El desarrollo de un sistema experto implica algo más que analizar el problema y resolverlo. Incluye la identificación de los expertos y otras fuentes de conocimiento en el dominio, la adquisición y la validación del conocimiento necesario, la identificación y la comprensión de los usuarios finales y su entorno, y la organización y la dirección del esfuerzo de desarrollo.

Un ejemplo concreto de algunas características del problema del sistema NEOMYCIN sería:

Dominio del Problema
Enfermedades infecciosas en la sangre
Conocimiento médico: organismos infecciosos relacionados con la historia del paciente, síntomas, y resultados de pruebas de laboratorios; tratamientos con drogas, métodos de diagnosis de problemas.

Problema a resolver dentro del dominio
Diagnosis y recomendación de terapia para pacientes hospitalizados con bacteremía, meningitis, y cistitis. También orientado a servir como conocimiento base para educar a estudiantes de medicina.

Fuentes de Conocimiento
Base de conocimiento existente en MYCIN; médicos especialistas; libros de medicina.

Entorno de la Aplicación
Usuarios finales: Especialistas médicos; estudiantes de medicina.

Son muy pocos los esfuerzos de desarrollo que están bien prediseñados. Existen casos en los que el dominio de la aplicación es muy complejo y el dominio del conocimiento ambiguo. Por lo tanto, sólo puede entenderse el problema después de haber realizado un importante esfuerzo en investigación y experimentación; esfuerzo que posiblemente implique el desarrollo de varias iteraciones de prototipado y diseño.

Este hecho puede derivar en una contradicción, ya que la selección de una herramienta sólo puede realizarse después de haber analizado las características del problema, y esto sólo es realizable tras haber experimentado y construido un prototipo del sistema experto, para lo cual es necesario una herramienta.

Dominio del Problema. El tipo de conocimiento y de procesamiento que caracteriza un dominio puede servir como criterio para seleccionar una herramienta. (Por ejemplo, la simulación militar podría requerir el razonamiento espacial; y los sistemas de control nucleares presentan restricciones de tiempo real.

Algunas herramientas incorporan mecanismos específicos y conocimiento orientado hacia un dominio particular. La disponibilidad de librerías de estadística, de gráficos, de acceso a bases de datos, etc.., también debe ser tenida en cuenta a la hora de decidir entre herramientas.

Problema a resolver dentro del dominio. El problema a resolver puede contener tipos especiales de conocimiento, de procesamiento o requisitos representacionales que pueden derivar en determinados requisitos en una herramienta. El problema establece también restricciones en la capacidad, rendimiento, y coste del sistema definitivo, así como en su disponibilidad, su fiabilidad, en la consistencia y en la posibilidad de mantenimiento.

Otros atributos a considerar serían: requisitos de almacenamiento y de complejidad que se esperan, restricciones operacionales como velocidad de ejecución, requisitos de tiempo real y compatibilidad, o la necesidad de una verificación formal.

Adquisición del conocimiento experto (Expertise). Es importante identificar cualquier característica o restricción que se pueda aplicar a las fuentes de conocimiento de área en cuestión, incluyendo la necesidad de múltiples fuentes o la coordinación de varias bases de conocimiento.

Entorno objetivo. Este determina el hardware necesario y la necesidad de su integración con el existente, junto con el software, bases de datos y redes.

También establece requisitos y restricciones en la capacidad, rendimiento y el coste de distribución del sistema objeto; así como su fiabilidad, consistencia y facilidad de

mantenimiento. Lo mismo sucede con las características de los usuarios finales, que determinan la interfase de usuario y los requisitos de justificación para el sistema definitivo, y por lo tanto para la herramienta.

2.2.2 Características del Proyecto

Incluye las características del proyecto del sistema experto, su entorno de desarrollo, y su equipo de desarrollo.

Extensión. Los factores más importantes a la hora de determinar qué tipo de sistema será construido (y con que herramienta será desarrollo) son su extensión, metas y presupuesto. En particular, la extensión de un proyecto determinará en que fases se centrará el mismo (e.j. prototipado, desarrollo, etc).

La mayoría de las herramientas de sistemas expertos proporcionan apoyo durante todo el desarrollo y distribución del sistema experto, pero un determinado proyecto puede que esté centrado sólo en algunas fases del proceso. Un proyecto que esté relacionado con el prototipado de un sistema necesitará una herramienta que ofrezca flexibilidad en la representación del conocimiento y facilidad para construir prototipos.

Por otro lado, si un proyecto ya ha definido su problema y tiene establecida una representación del conocimiento, entonces debe elegirse una herramienta que esté orientada a la construcción del sistema, no a las fases de adquisición o prototipado.

Muy relacionado con el concepto de extensión está el tema del presupuesto. Las herramientas más potentes permiten un mayor ahorro de esfuerzo y dinero a la hora de producir resultados, pero las herramientas también cuestan dinero y se necesita tiempo para poder conocerlas y utilizarlas eficientemente.

Entorno de Desarrollo. Este delimita el software y el hardware sobre el que la herramienta deberá correr, así como las interfases de red y de bases de datos que se deben proporcionar durante el desarrollo. Estos factores pueden ser impuestos o el proyecto puede designarlos.

Equipo de Desarrollo. Si el equipo humano es muy grande, el coste de instruir a los componentes del equipo en el uso de la herramienta puede ser prohibitivo. El tamaño del

equipo puede sugerir que la herramienta posea la capacidad de desarrollo cooperativo, bases de datos compartidas, etc.

También es importante la composición, preferencias, y la experiencia previa del equipo. De igual manera, las características del equipo de ingeniería del conocimiento influirá en el tipo de herramienta que soporte un método determinado de adquisición del conocimiento.

2.3 Capacidades de la Herramienta

Cuando se estudia la presencia de una característica específica en una herramienta, se corre el riesgo de que se pierda el contenido esencial de la misma. Por ejemplo, el utilizar una característica actual como backward chaining puede que se quede anticuado a corto plazo. Es importante centrarse en las capacidades que ofrece dicha característica (en este caso razonamiento orientado a metas), en lugar de en la característica misma. Las capacidades poseen un nivel semántico mayor que el que ofrecen las características.

Las herramientas que están actualmente en el mercado ofrecen muchas características que soportan gran cantidad de capacidades. Es importante centrarse en las capacidades de la herramienta, más que en las características se que ofrecen para satisfacerlas. Por ejemplo, una herramienta puede soportar la capacidad de jerarquías de objetos por medio de distintas características, como frames o reglas. Los usuarios normalmente necesitan de alguna capacidad sin preocuparles la característica que la implementa.

La siguiente tabla muestra algunos ejemplos de capacidades y de algunas características que las implementan. Esta lista no es ni mucho menos exhaustiva, y los ejemplos de las características son sólo ilustrativos. En las herramientas actuales pueden encontrarse muchas más capacidades y características que las soportan.

CAPACIDAD	EJEMPLOS DE CARACTERÍSTICAS
Adquisición del Conocimiento	Inducción de reglas, Ayudas a la construcción de modelos
Concurrencia	Procesamiento Distribuido // paralelo
Chequeo de Consistencia	Chequeo de la sintaxis de la base de conocimiento
Inferencia y control	Iteración, encadenamiento forward/ backward
Integración	Acceso a otros lenguajes

CAPACIDAD	EJEMPLOS DE CARACTERÍSTICAS
Interface E/S	Windows, gráficos, animaciones
Justificación	Traza, referencias cruzadas
Manejo de Incertidumbre	Factores de certeza, lógica difusa
Metaconocimiento	Reglas de control de inferencia, autoorganización de los datos
Optimización	Anticipación Inteligente, compilación de reglas
Procesamiento Aritmético	Operadores aritméticos, coma flotante extendida
Representación	Reglas, marcos, pizarras, redes semánticas.

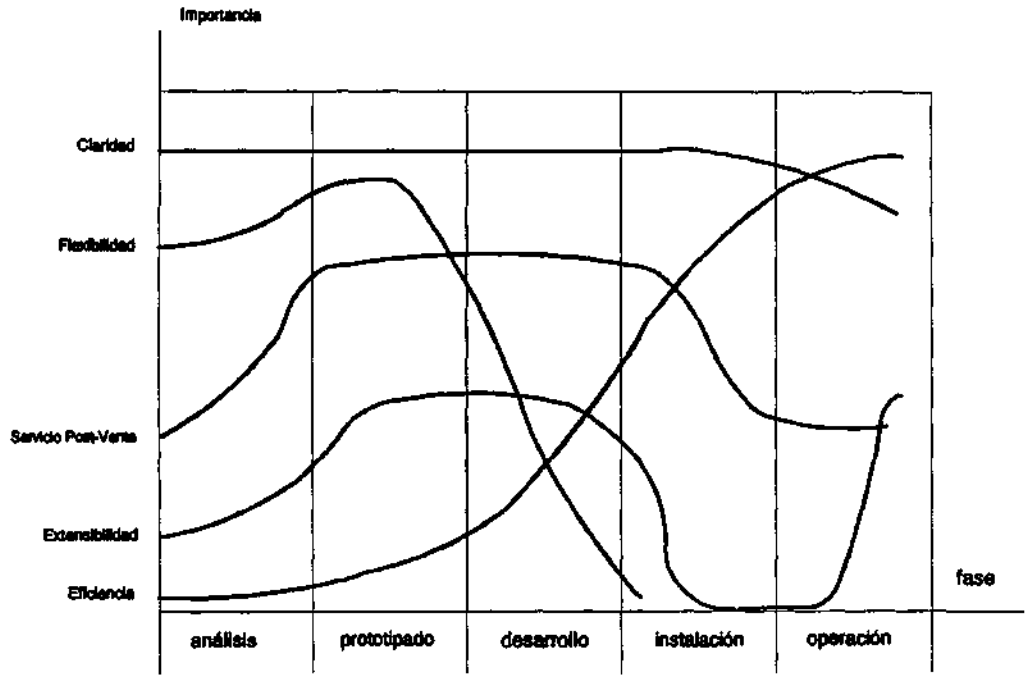
2.4 Métricas

Las métricas se aplican a capacidades particulares de una herramienta utilizando las técnicas de evaluación que serán descritas posteriormente. Las métricas que se muestran recogen las cualidades más importantes de la herramienta con un número relativamente pequeño de medidas. Cada métrica permite analizar en profundidad la herramienta mediante el estudio de las diferentes facetas (como la flexibilidad, eficiencia, o facilidad de uso) de sus capacidades, más que evaluarlas de acuerdo a una única medida.

Las métricas a considerar son: coste, flexibilidad, extensibilidad, claridad, eficiencia, y servicio post-venta

La siguiente figura muestra la importancia de las métricas a lo largo de las fases de desarrollo. El coste no se incluye en dicha figura ya que su comportamiento es sensiblemente diferente. El gráfico muestra las relaciones típicas entre métricas, aunque dichas relaciones puedan variar para proyectos diferentes. A continuación se comentarán estas métricas.

Coste. Incluye gastos ocultos tales como costes de integración y de entrenamiento, así como el precio de compra y de mantenimiento de una herramienta. El coste se puede definir en términos de dinero, personas, maquinaria, computación utilizada, tiempo, etc. Esta métrica es importante durante todo el ciclo de vida, pero para propósitos de evaluación su relevancia crece durante las transiciones entre fases.



Flexibilidad. Incluye la potencia representacional (estructuras de datos y mecanismos de razonamiento), conveniencia para la tarea en cuestión, espectro de aplicabilidad, y sofisticación. A medida que las decisiones de implementación se fijan, disminuye la importancia de la flexibilidad, llegando incluso a ser una factor negativo en las fases siguientes donde necesidad de un fácil mantenimiento conduce al requisito de que el sistema sea estable.

Extensibilidad. Incluye las posibilidades de aplicabilidad, de acceso a los parámetros del sistema, facilidad de integración, portabilidad, y posibilidades de ampliación. La extensibilidad alcanza su máxima importancia durante el diseño y la implementación, y decrece durante la instalación (fielding) donde el sistema ya debería ser más que estable. Su importancia podría aumentar otra vez en el caso de que el mantenimiento requiriese de nuevas ampliaciones o reintegraciones.

Claridad. Incluye la facilidad de comprender y de utilizar la herramienta, la eficiencia cognitiva (cuantos conceptos deben recordarse para usar la herramienta), posibilidad de mantenimiento, modularidad, facilidad de aprendizaje de la misma, coherencia de las características de la herramienta, y como es la respuesta de la herramienta. La claridad es

importante durante todas las fases, aunque su sacrificio puede justificarse una vez que el sistema es operacional, y sólo en el caso de que la necesidad de la eficiencia así lo requiriese.

Eficiencia. Incluye la velocidad de respuesta y utilización de los recursos computacionales y de memoria. La eficiencia se manifiesta durante el desarrollo en términos de velocidad de compilación, tiempo de respuesta, y requisitos de memoria de la base de conocimiento. Sin embargo, el generar un sistema eficiente se contradice con las posibilidades de eficiencia de la herramienta. La importancia de la eficiencia tiende a mantenerse baja hasta que el sistema es operativo.

Servicio post-venta. Incluye la filosofía de venta, la disponibilidad del sistema, la fiabilidad, portabilidad, y la consistencia del mismo. La importancia de esta métrica aumenta a medida que los usuarios empiezan a conocer la herramienta, y permanece alta hasta que el sistema definitivo haya sido distribuido.

2.5 Técnicas de Evaluación

Las técnicas a emplear con algunas de las métricas vistas son obvias y directas. La evaluación del coste inicial de una herramienta consistiría en preguntar su precio; sin embargo, la evaluación del coste de aprendizaje, o los costes a largo plazo dista mucho de ser trivial. De igual manera, la comprobación de características se realizaría preguntando al vendedor si la herramienta posee una cualidad determinada.

Las técnicas de evaluación que se aconsejan no intentan generar una valoración cuantitativa, sino producir resultados que necesitarán de una interpretación inteligente. Las técnicas de evaluación que ofrecen mejores resultados son:

- Comparación directa
- Benchmarks
- Entrevistas y cuestionarios
- Librería de estudios de casos y de esfuerzos de desarrollo
- Sistemas basados en el conocimiento para la evaluación de herramientas

Comparación Directa. La comparación directa entre herramientas puede ser muy valiosa, ya que estas deben centrarse en las respectivas capacidades de las herramientas y deben dejar claro de que forma dichas capacidades son diferentes o incomparables.

Benchmarks. En el contexto de sistemas expertos, no es en una medida cuantitativa del rendimiento como podría ser cuantas reglas puede procesar por segundo. Un benchmark consistiría en la formulación de un problema 'tipo' que puede ser pequeño o complejos. Los test pequeños no necesitan tener muchos datos asociados con ellos. Los benchmarks mayores pueden incluir información acerca del rendimiento y estadísticas (como el tiempo que necesitó para implementar una solución con una herramienta determinada), pero el énfasis se debe centrar en el contenido y en el estilo de la solución propuesta: el evaluador sería siempre responsable de la interpretación de cualquier dato numérico en términos de la calidad de la solución.

Los test pequeños pueden usarse para comparar capacidades específicas de las herramientas (E.j: como podría representar una jerarquía de clases), interpretándolas sobre la base del estilo de las soluciones más que en su cumplimiento. La clave del éxito de los benchmarks es formular problemas que sean específicos pero que no necesiten de una implementación particular.

Técnicas Adicionales. Entrevistas, cuestionarios y consejos personales de otros diseñadores y colegas pueden proporcionar información muy valiosa, la cual es a menudo más fiable que los resultados de estudios de evaluación formales. Una librería de estudios de casos y de intentos de desarrollo de sistemas expertos pueden ser una fuente muy valiosa de datos para la evaluación siempre que verifiquen que:

- Los datos sean suficientes como para ser representativos.
- Los datos se mantengan actualizados según evolucionan las herramientas y experimentan cambios.
- Los datos sean accesibles a un gran número de usuarios potenciales de herramientas y estén clasificados de acuerdo a las características de las aplicaciones.

Un sistema basado en el conocimiento para la evaluación de herramientas podría ayudar a sintetizar una decisión apoyándose en los resultados de la evaluación de varias herramientas de acuerdo a varios criterios.

2.6 Contextos

Esta dimensión representa el contexto en el que puede usarse una herramienta. Cada contexto coincide con la fase de desarrollo en la que es dominante, aunque un determinado contexto puede aplicarse durante varios estados de desarrollo: por ejemplo, los requisitos de la herramienta para la 'operación' pueden establecerse durante el desarrollo conceptual de un proyecto. La representación de estos conceptos como dimensiones separadas enfatiza esta independencia y permite aplicar los temas de cada contexto a todos los estados del desarrollo. También debe tenerse un especial cuidado con las transiciones entre las fases de desarrollo, las cuales son pueden ser tan importantes como las mismas fases.

Los contextos potenciales (y las fases de desarrollo) son: Conceptualización y análisis, prototipado, desarrollo, instalación (fielding), operación y mantenimiento

Conceptualización. Enfatiza la posibilidad que tiene una herramienta para la conceptualización, la formalización y la descomposición de un problema, y para identificar y organizar conceptos clave y definir el ámbito del problema. Esto implicaría la exploración de soluciones y representaciones alternativas así como un diseño conceptual.

Prototipado. Enfatiza las facilidades de la herramienta para el desarrollo rápido, permitiendo diferentes aproximaciones y representaciones, y la posibilidad de intentar rápidamente implementaciones alternativas.

Desarrollo. Considera una herramienta según se utiliza para desarrollo un sistema experto. Se enfatizan las posibilidades de desarrollo del software (incluyendo facilidades de depuración, gestión de la configuración, etc).

Instalación. Se centra en las facilidades de la herramienta a la hora de instalar el sistema definitivo en su entorno de trabajo.

Operación/Mantenimiento. Son las capacidades que tiene la herramienta de mejorar el rendimiento, las posibilidades de un servicio de mantenimiento, y el apoyo al sistema experto generado en su entorno definitivo.

3 Metodología

El objetivo de esta metodología es el de evitar prejuicios e intimidaciones a la hora de validar una herramienta con la estructura vista anteriormente. Los pasos de esta metodología son:

1. Determinar las características de la aplicación.
2. Identificar los contextos relevantes.
3. Derivar las capacidades significativas de la herramienta.
4. Identificar las métricas discriminantes y las técnicas de evaluación.
5. Identificar las herramientas disponibles.
6. Filtrar las herramientas disponibles para identificar las herramientas candidatas.
7. Podar y priorizar cada dimensión de la estructura.
8. Aplicar el esquema de la estructura para evaluar y seleccionar las herramientas.

Antes de desarrollar los puntos anteriores convendría responder a algunas cuestiones. La validez del proceso de evaluación podría ser discutible en el caso de que alguna de las preguntas tuviera una respuesta negativa.

(1) ¿ Tiene la evaluación un usuario identificado ?

Las evaluaciones dirigidas a los managers, a los directores técnicos o a los usuarios finales poseen requisitos diferentes. El tener este factor en cuenta podrá evitar el producir resultados superfluos o mal dirigidos.

(2) ¿ Será la evaluación imparcial y objetiva ?

Es importante preguntarse si se puede esperar que los que realizan la evaluación vayan a ser imparciales. Para que pueda tener alguna credibilidad, la evaluación debe ser imparcial. esto significa que los algoritmos de evaluación, las prioridades y los pesos de importancia se establezcan por anticipado, y si fuera posible, por expertos en el dominio o por otras partes

no implicadas en el proyecto. Una vez que se han establecido los pesos, deben considerarse como inmodificables a menos que se descubra que son insostenibles, en cuyo caso deberán ser redefinidos.

(3) ¿ Cómo se controlará la precisión y la parcialidad de la evaluación ?

En otras palabras, ¿ Quién evalúa al evaluador ?. Esto es necesario para evitar posibles errores y parcialidades.

(4) ¿ La evaluación será válida indefinidamente ?

El proceso de evaluación debe estar diseñado para alcanzar sus conclusiones a tiempo para ser usadas. Debido a que las herramientas evolucionan rápidamente, los resultados de la evaluación pueden volverse rápidamente obsoletos a menos que se tomen algunas precauciones para evitarlo. Es muy importante especificar las versiones y las fechas de venta de las herramientas evaluadas.

3.1 Los ocho pasos para la evaluación

En esta sección examinaremos los ocho pasos vistos anteriormente.

Paso 1: Determinar las características de la aplicación.

Aunque el dominio de la aplicación y del problema pueda ser difícil de especificar, es necesario determinar las características de la aplicación lo antes posible, puesto que las capacidades de la herramienta son deducidas de dichas características. También es importante determinar las características del proyecto (metas, ámbito y presupuesto), y especificar el equipo y el entorno de desarrollo.

Las características de la aplicación deberán ser pesadas según sea su certeza y su importancia. El ámbito y las metas del proyecto determinarán que requisitos son obligatorios y cuales son negociables.

Paso 2: Identificar los contextos relevantes.

Los contextos en los que se centrará el proyecto proporciona el otro factor principal para establecer las capacidades que deberá tener la herramienta. Si el propósito del proyecto es la fase de prototipado puede que en el caso de que el proceso de desarrollo se extendiera a las fases siguientes la herramienta fuera incapaz de trabajar correctamente.

Paso 3: Derivar las capacidades significativas de la herramienta.

Las capacidades de la herramienta se deducen de las características de la aplicación y de los contextos relevantes. Es importante el realizar un pesado de las capacidades sobre una escala desde imprescindible a deseado, para usar en el filtrado de las herramientas disponibles (paso 6). Estos pesos deben ser derivados de los pesos de las características de la aplicación establecidos en el paso 1.

Paso 4: Identificar las métricas discriminantes y las técnicas de evaluación.

Se dará el caso en el que determinadas métricas, como el coste, tendrá un alto valor determinante a la hora de seleccionar una herramienta. También se debe establecer la mejor técnica de evaluación disponible para aplicarla en este estado de la evaluación. Si no existiera una métrica con este poder de veto para un proyecto, este paso no tendría efecto.

Paso 5: Identificar las herramientas disponibles.

Este paso implica un filtro implícito, ya que es difícil encontrar todas las herramientas disponibles.

Paso 6: Filtrar las herramientas disponibles para identificar a las herramientas candidatas.

Utilizando las capacidades requeridas derivadas en el paso 3 y la métrica discriminante (si existe) identificada en el paso 4, se filtran las herramientas disponibles para producir una conjunto de herramientas candidatas a ser evaluadas con más detalle.

Paso 7: Podar y priorizar cada una de las dimensiones.

Se debe podar cada dimensión para eliminar criterios irrelevantes o inaplicables, y los términos restantes deberán ser priorizados o pesados. En este punto debe conocerse lo suficiente acerca de las características de la aplicación, de los contextos, de las capacidades de las herramientas disponibles para priorizar cada dimensión por separado. Por ejemplo, si ya se ha aplicado el filtro del coste, resultando un conjunto de herramientas cuyo precio es similar, el darle ahora a la métrica coste una prioridad alta no sería efectivo.

Cada dimensión debe ser priorizada en términos de la importancia esperada y de la relevancia de cada componente en esa dimensión. La excepción es la dimensión de técnicas de evaluación; estas técnicas deben priorizarse en términos de su disponibilidad, aplicabilidad, coste, y duración de la validez de su aplicación.

Paso 8: Aplicar el esquema de la estructura para evaluar y seleccionar herramientas.

Finalmente, se aplican las dimensiones priorizadas a las herramientas candidatas utilizando el esquema de la página 3. Es decir, se utilizan las apropiadas técnicas de evaluación para evaluar las métricas relevantes aplicadas a cada capacidad de una herramienta particular en un contexto determinado, dadas unas características de la aplicación determinadas. Este paso podría necesitar un gran número de evaluaciones individuales, pero esto es inevitable.

La credibilidad de una evaluación depende de como esté hecha formalmente y objetivamente. Esta metodología no intenta indicar a los analistas cómo interpretar los resultados de sus evaluaciones, sino sólo producir datos que les ayuden a tomar sus decisiones.

4 Conclusiones

Debido a que la terminología del campo de sistemas expertos evoluciona rápidamente, comparar características de herramientas diferentes es de poca utilidad. En lugar de ello deben analizarse, compararse y evaluarse las capacidades proporcionadas por esas características. La estructura presentada muestra como usar determinadas técnicas de evaluación para aplicar determinadas métricas a determinadas capacidades de una herramienta para una aplicación determinada en un contexto determinado.

Aunque la dificultad de la comparación y de la selección de herramientas podría desanimar a un analista que se enfrenta con un decisión, esta dificultad es consecuencia de la riqueza del campo y en el que nuevas ideas se están incorporando en las herramientas. La aproximación de evaluación que se propone no es una respuesta definitiva a un problema establecido, sino una estrategia para gestionar un problema dinámico cuyo impacto sobre la ingeniería del software sólo se está empezando a sentir.

Bibliografía

Guida, G.. Tasso, C.. Topics in expert system design. ed. North-Holland. 1990

CAPÍTULO II

Nexpert Object v. 2.0

Índice

1. Introducción 1

2. Elementos de Nexpert 3

3. Descripción funcional 5
   - 3.1 Presentación 5
   - 3.2 Estructuras de representación de objetos 6
   - 3.3 Estructuras de representación del conocimiento 8
   - 3.4 Estructuras de representación dinámicas 10

4. Mecanismos de herencia 11
   - 4.1 Herencia de propiedades 11
   - 4.2 Herencia de valores 12
   - 4.3 Herencia de meta-slots 13
   - 4.4 Estrategias de herencia 15

5. Interpretación y comparación con modelos 19
   - 5.1 Interpretación 19
   - 5.2 Comparación con modelos (Pattern matching) 19

6. Proceso del conocimiento 23
   - 6.1 Procesamiento de reglas 24
   - 6.2 Mecanismos de inferencia 24
   - 6.3 Estrategias de inferencia: resolución de conflictos 29
   - 6.4 Inferencia no-monotónica 29

Apéndices

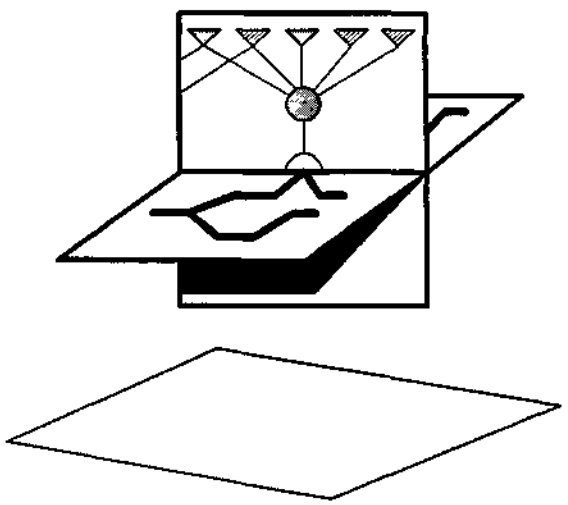
   - A. Glosario de términos 33
   - B. Menús en Nexpert 35

1 Introducción

Dado que los sistemas expertos normalmente contienen conocimiento sobre los objetos del mundo real, las herramientas destinadas a tal fin deben contener mecanismos que ayuden al ingeniero del conocimiento a representar el conocimiento de estos objetos. **Nexpert Object** se caracteriza por su capacidad de representar un *mundo* de objetos sobre el cual actuará el conocimiento.

En Nexpert, el conocimiento básicamente se expresa mediante reglas. Una aplicación desarrollada con esta herramienta puede limitarse a usar solamente reglas o bien incorporar otro mecanismo de representación: los objetos.

La figura 1 ilustra la integración de reglas y objetos en Nexpert; se han representado las dos dimensiones de Nexpert Object. El plano horizontal contiene las reglas y el plano vertical contiene las estructuras de objetos. El punto de intersección representa el punto de actuación en ese momento, el *centro de atención* (focus of attention).



Logotipo de Nexpert Object
Figura 1

2 Elementos de Nexpert

La herramienta Nexpert Object incorpora los siguientes componentes en el entorno de desarrollo:

- **Interface de usuario gráfica (GUI).**
  Nexpert está desarrollado sobre el entorno gráfico OpenWindows, el cual se caracteriza por el uso de ventanas, iconos, punteros... Esta interface gráfica tiene la ventaja de estar estandarizada, y tener la misma apariencia independientemente del hardware sobre el que esté ejecutándose. Las características de una interface gráfica ayudan al usuario en la construcción y depuración de la aplicación.

- **Herramientas de razonamiento y representación.**
  Es el núcleo *(kernel)* de Nexpert en forma de funciones y algoritmos, y que evidentemente se utiliza para ejecutar las aplicaciones basadas en el conocimiento. Además estas funciones se distribuyen en forma de librerías "C" y permiten desde el acceso a la representación de los objetos hasta la interacción con los mecanismos de inferencia.

- **Interface para programar aplicaciones (API).**
  Esta interface permite al usario llamar desde Nexpert a rutinas externas y programas escritos en lenguajes de programación estandar (C, Pascal, Fortran...). También permite, que programas de aplicaciones a gran escala, puedan combinar facilidades de inteligencia artificial con las de programación clásica.

Los usuarios, a la hora de desarrollar una aplicación, pueden usar indistintamente la interface de aplicaciones (API) y la interface gráfica (GUI).

3 Descripción funcional

3.1 Presentación

En la literatura, se suele hablar de Nexpert Object como un sistema híbrido en el ámbito de las herramientas de sistemas expertos. Esto es debido a que utiliza dos mecanismos de representación de conocimiento diferentes.

Por un lado y para permitir los mecanismos de razonamiento, Nexpert Object utiliza reglas. Estas son las estructuras de conocimiento más básicas que nos encontramos en esta herramienta. Mediante estas reglas, el sistema realiza el proceso de conocimiento utilizando técnicas como encadenamientos hacia atrás o hacia delante[1], para obtener las conclusiones solicitadas.

Por otro lado, el dominio sobre el que operan las reglas tiene una variada gama de estructuras de representación. El *mundo* es modelado en términos de objetos, clases y propiedades. Los valores de las propiedades específicas de los objetos y clases se almacenan en los *slots*. Son las variables de Nexpert, en ellos se almacena toda la información que se recolecta del entorno.

Además, los slots poseen *meta-slots,* los cuales describen el comportamiento de los primeros. Las posibilidades que ofrecen los meta-slots van desde la definición del mensaje que se le presenta al usuario, para preguntarles el valor del slot *(prompt),* hasta los métodos que se deben seguir en la búsqueda del valor de un parámetro (recuperarlo de una base de datos, ejecutar una rutina externa...).

También existe la posibilidad de crear objetos y clases dinámicamente, así como crear las relaciones de herencia entre ellos. De esta forma un objeto puede *nacer* como hijo de una clase y *morir* como hijo de otra.

Existen otras facilidades a incluir como parte operativa dentro de las reglas, éstas son el *pattern-matching* y la *interpretación*, que permiten referenciar y comparar objetos y clases.

[1] Estas técnicas se verán en detalle en próximos apartados.

Por último, destacar que Nexpert soporta mecanismos de herencia múltiple entre objetos y clases. Estos mecanimos permiten la herencia (de padres a hijos o viceversa) de propiedades, valores y algunos meta-slots.

3.2 Estructuras de representación de objetos

La herramienta Nexpert Object incorpora las siguientes estructuras de representación de objetos:

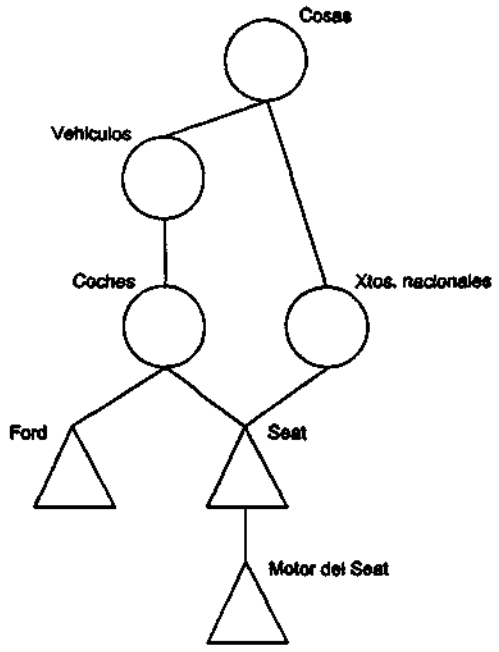
- **Objeto.**
  Es la entidad más básica de información. Se representa mediante un triángulo.

- **Clase.**
  Es la generalización de un conjunto de objetos. Se representa mediante un círculo. Un objeto puede pertenecer a varias clases.

- **Subclase.**
  Es la especialización de una clase. Las clases pueden tener cualquier número de subclases y crear una jerarquía de clases con el número de niveles que se desee. Las subclases heredan todas las *propiedades* de la clase a la que pertenecen.

- **Subobjeto.**
  Es otro nivel más dentro de la jerarquía de clases y objetos. Los subobjetos se diferencian de las subclases, en que no heredan las propiedades del objeto al que pertenecen. Se utilizan para representar relaciones del tipo *es parte de*.

La figura 2 ilustra las diferentes estructuras de representación anteriormente descritas.



Jerarquía de clases y objetos
Figura 2

- **Propiedad.**

  Son los tipos de datos de un objeto o una clase, y como su propio nombre indica definen las propiedades del objeto (o la clase). Pueden ser multievaluadas. Se representan mediante un rectángulo.

  Pueden haber objetos o clases sin propiedades definidas explícitamente. Esto se debe a que, por defecto, todos los objetos o clases tienen definidos una propiedad llamada *Value*, la cual puede ser del tipo de dato que se quiera.

  Las propiedades son globales a toda la base de conocimiento, es decir, no puede haber una propiedad definida en un objeto con un tipo de dato y la misma propiedad definida en otro objeto con un tipo de dato diferente. De esta forma se asegura que un objeto que pertenezca a dos clases diferentes, al heredar las propiedades, no se encuentre con inconsistencias.

- **Slot.**

  Su función es almacenar el valor de la propiedad (de un objeto o una clase). Se representan mediante un cuadrado.

  Para acceder al valor almacenado dentro del slot de un objeto o clase, la sintaxis a seguir es la siguiente:

  ```
  ( CLASE | OBJETO ) . PROPIEDAD.
  ```

Ejemplos.
- *coche.velocidad* es la velocidad del coche.
- *ordenador.cpu* es la cpu que tiene el ordenador.
- *examen.Value* (o *examen* indistintamente) se refiere al slot que, por defecto, tienen todos los objetos y clases.

Los estados posibles de almacenamiento de un slot son:

ESTADO	SITUACION
UNKNOWN	Aun no se ha evaluado el slot
Valor asignado	Evaluado y conocido su valor
NOTKNOWN	Se evaluó pero se desconoce su valor

- **Meta-slots.**
  Describen el comportamiento de un slot a la hora de ser evaluado. Estas características incluyen:

  - Prompt line. Cómo pregunta el sistema por el valor de un slot cuando éste es requerido
  - Prioridades de inferencia y herencia.
    - Inference priorities. Código numérico que define la prioridad de un slot sobre otro a la hora de ser evaluado.
    - Inheritance priorities. Código numérico que define la prioridad de un slot sobre otro a la hora de poner su valor a sus *herederos*.
    - Inheritability settings. ¿Deben heredarse los slots de padres a hijos, o de hijos a padres? ¿Deben heredarse los valores?.
    - Inheritance strategy. La estrategia de herencia de valores que se va a usar, en la cual se determina si ésta se hace primero en profundidad o amplitud, y si los objetos propagan sus valores a sus hijos con preferencia a las clases de su mismo nivel.
  - Order of sources. Cuáles son las fuentes que se deben consultar para obtener el valor del slot (p.e. recuperar de una base de datos, ejecutar una rutina externa...). Cuando Nexpert requiere el valor de un slot ejecuta secuencialmente todos los procedimientos que definamos mientras el valor sea *Unknown*.
  - If change. Qué hacer si cambian los valores del slot. A esta característica, también se la conoce como *demonios*.

3.3 Estructuras de representación del conocimiento

La herramienta Nexpert Object incorpora las siguientes estructuras de representación del conocimiento:

- **Reglas.**

  Es la estructura de datos que se utiliza en esta herramienta para desencadenar el razonamiento. En la estructura de la misma se observan tres partes bien diferenciadas:

Ejemplo

If LHS **Then** Hipótesis
Do RHS

LHS	Hipótesis
	RHS

- **Condiciones (LHS).** Son las restricciones que se deben cumplir para validar y ejecutar el consecuente de la regla. La estructura de la condición es:

  Operador + Expresión + Constante.

  Pueden haber varias condiciones, en cuyo caso estarán unidas por el operador lógico *and*. El orden de evaluación de las mismas va en función de las prioridades de inferencia (ver *Inference priorities*, pag.8).
- **Hipótesis.** El resultado de la evaluación de las condiciones de una regla da lugar a un valor booleano que es la hipótesis.

  Una misma hipótesis puede estar en el consecuente de varias reglas, por lo que el valor de la misma se obtendrá por la validación de cualquiera de las reglas.

  De la misma forma una hipótesis puede aparecer en el antecedente de otras reglas, lo cual tiene como resultado un proceso de encadenamiento de éstas para obtener el valor de una hipótesis (ver *Backward chaininig*, pag.25). Este es el principal proceso de razonamiento que implementa Nexpert.
- **Acciones.** Son las acciones que se desencadenan tras haberse validado las condiciones y por tanto la hipótesis. La estructura de una acción es:

  Operador + Expresión + Constante.

Ejemplo.
Si un un coche tiene 4 puertas y tiene baca, entonces se trata de un coche familiar y se desea que muestre un gráfico con un prototipo de coche familiar:

```
If coche.puertas=4 and Yes baca  Then familiar
                                 Do show "familiar" @KEEP=TRUE
```

3.4 Estructuras de representación dinámicas

Nexpert incorpora la posibilidad de crear objetos y links dinámicamente. Con esto se consigue mayor flexibilidad y acoplabilidad a programas, bases de datos....

Hay dos aspectos diferentes que se consideran en el mantenimiento de las estructuras dinámicas: la estructura en sí y las relaciones de herencia que se le asignan.

- Por un lado lado, se pueden crear *objetos dinámicos*. Para ello se proporciona el operador *CreateObject*, el cual puede crear un objeto sin propiedades y sin padres.

- Por otro lado, se pueden crear *links dinámicos*, que es la forma de dotar de propiedades, slots y meta-slots a uno de estos objetos. El operador que se utiliza es el mismo, *CreateObject*.
  El link puede ser creado tanto a entidades estáticas como dinámicas. Los mecanismos de herencia funcionan exactamente igual que en las estructuras estáticas (herencia de valores, propiedades y meta-slots).

Tantos los links como los objetos pueden ser borrados o reasignados.

4 Mecanismos de herencia

Los mecanismos de representación son muy útiles a la hora de estructurar el mundo que deseamos modelar, pero son realmente los mecanismos de herencia los que aportan consistencia a esta forma de representación. Hay tres tipos de herencia:

- **Herencia de propiedades.**
  Permite que se hereden propiedades de padres a hijos o viceversa, con lo cual se asegura la consistencia en la representación del mundo de objetos y clases.

- **Herencia de valores.**
  Permite que se hereden valores entre las clases y objetos que definan relaciones de cualquier tipo, con el fin de facilitar los mecanismos de evaluación de slots.

- **Herencia de meta-slots.**
  Hace posible la herencia de dos características que se definen en los meta-slots: *order of sources* e *if change*.

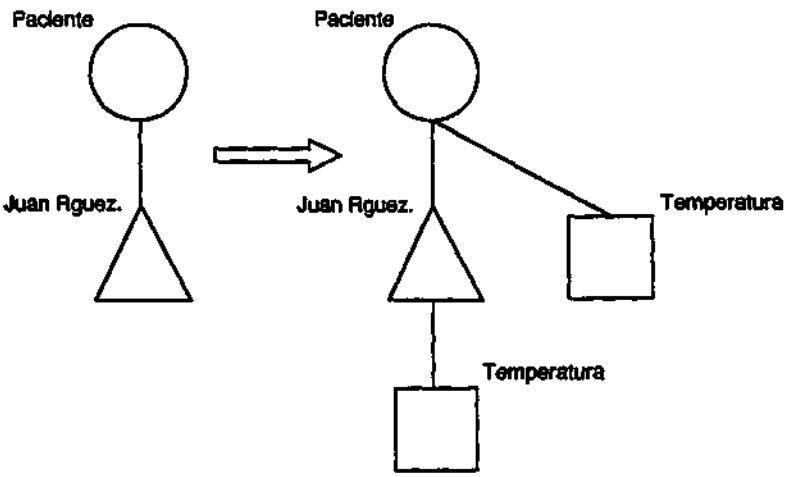
4.1 Herencia de propiedades

Se refiere a la capacidad de un objeto o subclase de heredar una propiedad particular de una clase. La norma que rige a este tipo de herencia es la siguiente:

- La herencia de propiedades por defecto se realiza de clases a subclases y de clases a objetos, pero no de objetos a objetos.
- La herencia es inmediata. Esto significa que todas las clases u objetos que estén relacionados, heredan automáticamente todas las propiedades. Incluso una nueva entidad que se creara posteriormente heredaría estas propiedades.
- En la estrategia por defecto se heredan las propiedades sólo descendentemente, aunque esto puede ser modificado (ver el apartado *estrategias de herencia*).
- La propagación es recursiva, es decir, la herencia de propiedades se realiza hasta llegar al último nivel o bien a un nivel en el que ya exista esa propiedad.

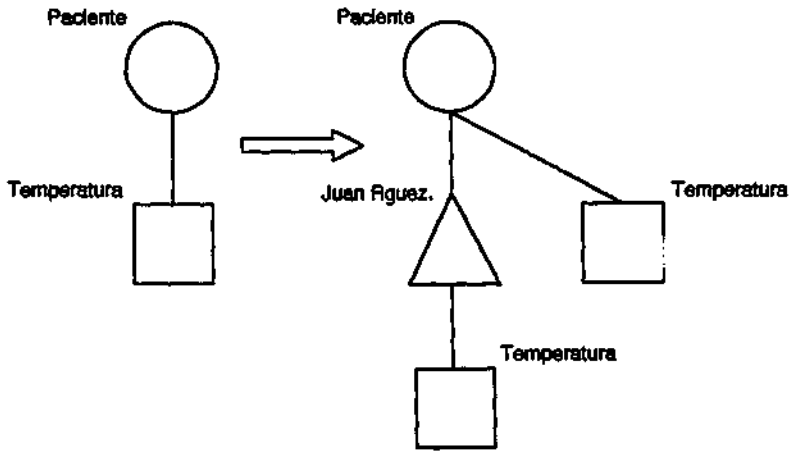
- Se heredan todos los slots, excepto el *.Value*.

Ejemplo 1.



Herencia de propiedades cuando se añade una propiedad
Figura 3

Ejemplo 2.



Herencia de propiedades cuando se crea un nuevo objeto
Figura 4

4.2 Herencia de valores

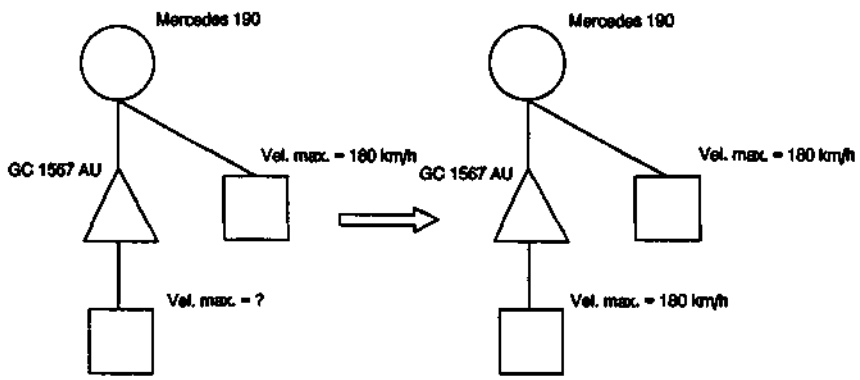
Es la capacidad de un slot para tomar el valor de su padre (o hijo) si su propio valor es *unknown*. Las características de este tipo de herencia son:

- A diferencia de la herencia de propiedades que es automática, la herencia del valor sólo se realiza en el momento en que se requiere.
- Se heredan todos los slots, excepto el *.Value*.
- No es recursiva, sólo afecta a los padres e hijos.
- Los valores pueden heredarse de objetos a subobjetos.

En la figura 5 se ilustra un ejemplo del funcionamiento de la herencia de valores entre clases y objetos.

Para entender exactamente como se realiza la evaluación de un slot, se recomienda ver el apartado de *herencia de meta-slots*.

Ejemplo.



Herencia de valores
Figura 5

4.3 Herencia de meta-slots

La herencia de meta-slots sólo opera sobre las características de *order of sources* e *if change*. El sentido de herencia es siempre descendente (no puede ser nunca hacia arriba) y puede ser entre clase y subclase, entre clase y objeto o bien entre objeto y subobjeto. No obstante, esta estrategia por defecto se puede modificar tanto globalmente como localmente (ver el apartado *estrategias de herencia*).

Cuando Nexpert requiere evaluar un slot, el algoritmo que sigue es el siguiente:

si el slot posee las acciones OS **entonces** se ejecutan estas acciones
sino
  si el slot del padre posee las acciones OS **entonces** las ejecuta
  sino se ejecuta el OS por defecto (Backward, InheritValueDown, InheritValueUp)

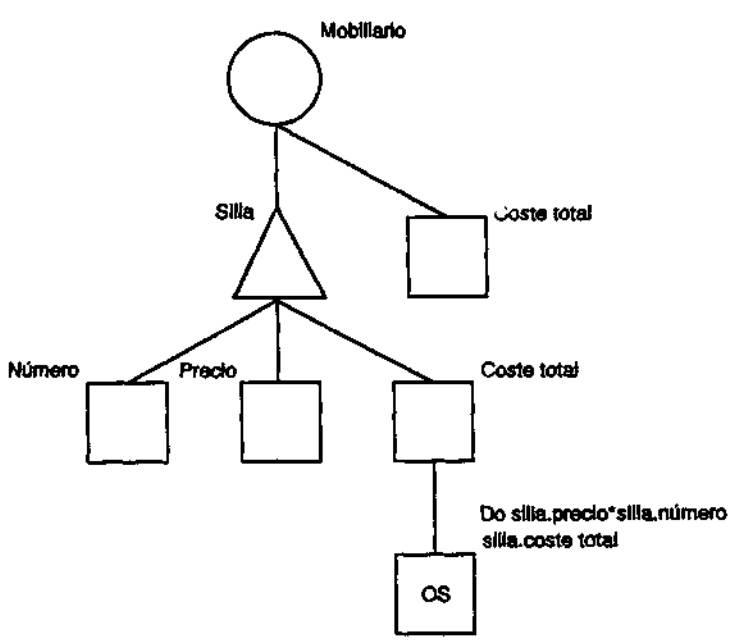
si el valor del slot sigue siendo Unknown **entonces** se le hace una pregunta al usuario.

De la misma forma, si el valor de un slot se modifica en el transcurso de una sesión de conocimiento se procede de la siguiente forma:

si las acciones IC están declaradas **entonces** se ejecutan
sino
  si el padre posee estas acciones **entonces** las ejecuta
  sino no se realiza ninguna acción, ya que por defecto no se realiza ninguna acción cuando se modifica algún valor.

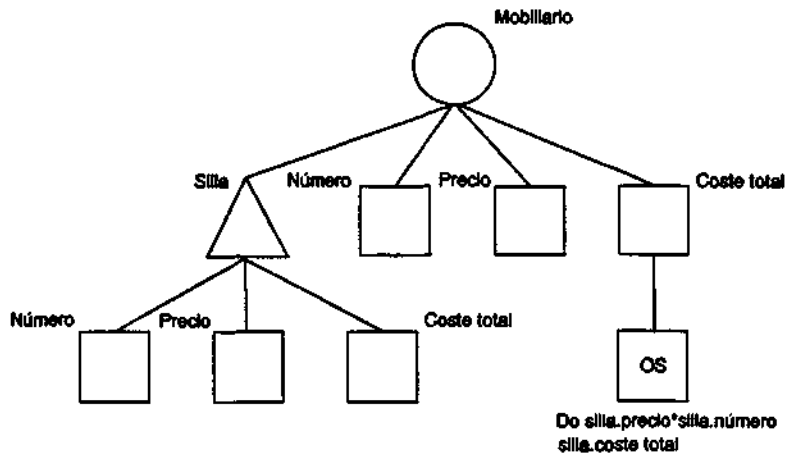
Dado que estos dos tipos de acciones se pueden heredar en cualquier momento, hay que tener en cuenta posibles particularidades en las acciones:

- Supongamos el siguiente ejemplo, la clase *mobiliario* con un objeto que pertenece a dicha clase, *Silla*. Si Nexpert necesitara calcular el slot *coste total* utilizaría las acciones OS que tuviera declaradas. La figura 6 ilustra este ejemplo.



Order of Sources (OS)
Figura 6

Ahora bien, si heredase las acciones de su padre, no habría posibilidad de realizar esta misma operación si no se dispusiese de la variable genérica *Self* (figura 7).



Uso de la variable Self
Figura 7

- También se pueden definir acciones (tanto en OS como en IC) que individualmente no se hereden a sus hijos. Esto se hace colocando un asterisco (*) como prefijo del operador de la acción en concreto.

ORDER OF SOURCES
* accion1
accion2
* accion3
* accion4

4.4 Estrategias de herencia

Hay tres tipos de estrategias de herencia diferentes, cada una de ellas relacionada con aspectos diferentes de la herencia:

- **Inheritability down** (Heredabilidad de padres a hijos). Se refiere a la capacidad de objetos y clases para heredar propiedades o valores de sus padres. Los valores por defecto para este tipo de estrategia son:
  - Herencia de propiedades de clases a subclases ON

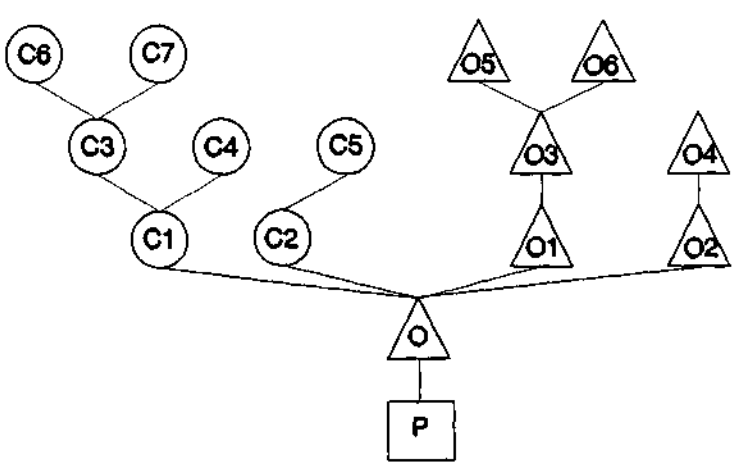
- Herencia de propiedades de objetos a subobjetos OFF
- Herencia de valores de clases a subclases u objetos ON
- Herencia de meta-slots (no modificable) OFF

- **Inheritability up** (Heredabilidad de hijos a padres). Se refiere a la capacidad de objetos y clases para heredar propiedades o valores de sus hijos. Los valores por defecto para este tipo de estrategia son:
  - Herencia de propiedades de subclases a clases OFF
  - Herencia de propiedades de subobjetos a objetos OFF
  - Herencia de valores de clases a objetos a subclases OFF

- **Resolución de conflictos** (Conflict Resolution). Cuando un objeto tiene varios padres que declaran la misma propiedad es necesario definir de cual de ellos obtiene el valor. Hay cuatro posibilidades entre las que elegir:

  - Class-first, breadth-first (por defecto)
  - Class-first, depth-first
  - Object-first, breadth-first
  - Object-first, depth-first

Ejemplo.



Resolución de conflictos
Figura 8

Sobre esta figura se pueden analizar las diferencias que existen entre ellas. Ante la necesidad de obtener el valor de la propiedad P, el orden de búsqueda del valor dentro del árbol de clases y objetos es el siguiente:

- *Class-first, breadth-first.* (se busca primero en las clases y en amplitud) C1, C2, O1, O2, C3, C4, C5, O3, O4, C6 C7, O5 y O6, hasta que se encuentre el valor.
- *Class-first, depth first.* (se busca primero en las clases y en profundidad) C1, C3, C6, C7, C4 ,C2, C5, O1, O3, O5, O6, O2 y O4.
- *Object-first, breadth-first.* (se busca primero en los objetos y en amplitud) O1, O2, C1, C2, O3, O4, C3, C4, C5, O5, O6, C6 y C7.
- *Object-first, depth-first.* (se busca primero en las clases y en profundidad) O1, O3, O5, O6, O2, O4, C1, C3, C6, C7, C4, C2 y C5.

Estos valores pueden modificarse:

- **Globalmente.** Desde la opción de menú *Strategy Dialog Box*.
- **Localmente**. Editando el Meta-slot correspondiente al slot de la propiedad en cuestión
- **En tiempo de ejecución.** Mediante el operador *Strategy*.

5 Interpretación y pattern matching

5.1 Interpretación

El valor del slot se interpreta para obtener el nombre de un objeto o de una clase. Consiste en averiguar este valor para sustituirlo literalmente en la expresión. Hay dos tipos de interpretación:

- **Sustitución por nombre de slot.**
  El resultado de la sustitución es el nombre de otro slot.

Ejemplos.

- \coche.tipo\.velocidad
  coche.tipo = "porsche" → porsche.velocidad

- 'disco_'\d.nombre\.espacio
  d.nombre = "duro" → disco_duro.espacio

- **Sustitución por nombre de string.**
  El resultado de la transformación es una string.

Ejemplos.

- @v(file.name)
  file.name = pepe.pas → "pepe.pas"

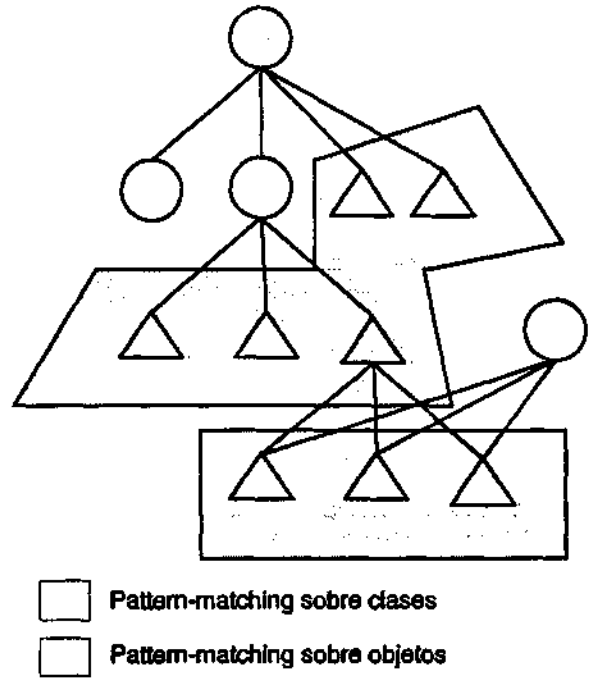
- "Hay @v(n.nodos) nodos..."
  n.nodos = 25 → "Hay 25 nodos..."

5.2 Comparación con modelos (pattern-matching)

La comparación con modelos o *pattern-matching* es otro método que permite la evaluación de los valores de los slots sin mencionarlos explícitamente. La principal característica de este servicio de Nexpert, es que permite referenciar múltiples objetos simultáneamente.

Internamente Nexpert crea una lista formada exclusivamente por objetos, sobre los cuales se podrán evaluar condiciones o realizar acciones.

Ejemplo.



Pattern-matching
Figura 9

Para hacer uso de este servicio Nexpert facilita dos operadores, los cuales actúan sobre una clase o un objeto. En la lista se incluyen todos los objetos a partir de la entidad (clase u objeto) sobre la que actúa el operador, hasta alcanzar el primer nivel de objetos:

- **Cualificador universal → { ... }**
  Permite seleccionar conjuntos de objetos y evaluar condiciones (LHS) que se cumplan en todos ellos. También se permite hacer operaciones (RHS) sobre todos ellos a la vez. Los caracteres reservados para representar este cualificador son { ... }.

- **Cualificador existencial → < ... >**
  Permite seleccionar conjuntos de objetos y evaluar condiciones (LHS) que se cumplan en alguno de ellos. Cuando se trata de realizar acciones (RHS), éstas se realizan sobre todos los objetos. Los símbolos que se usan para representar este cualificador son < ... >.

La lista creada mediante esta comparación es local a la regla en la que se haya usado dicha comparación. Una vez la hipótesis de la regla es evaluada y las acciones ejecutadas, la lista desaparece. En el caso de que los meta-slots OS e IC estuvieran definidos, la lista no desaparecería hasta que estas acciones se ejecutaran.

También es posible manejar más de una lista en el ámbito de una misma regla. La forma de hacer esto es duplicando los símbolos de los cualificadores. Por ejemplo, para referirnos a la lista uno utilizaríamos los símbolos <..> y {..}, para la segunda lista <<..>> y {{..}}, para la tercera <<<..>>> y {{{..}}}, y así sucesivamente.

Ejemplo.

> <ruedas>.presion 10	hipótesis
Yes {ruedas}.cadenas	Do "OK" {ruedas}.estado

La primera condición genera una lista con todas aquellas "ruedas" que tengan la presión mayor que 10. Si no hubiera ninguna "rueda" que cumpliera esta restricción, el resultado de la condición sería *false*.
La segunda condición opera sobre la primera lista generada, comprobando que todos los elementos de esa lista cumplan *tener cadenas*. Si no lo cumplieran todos los objetos la condición se evaluaría a *false*.
Notar que en este caso la evaluación de las condiciones no es conmutativa.
Por último, la acción generada operaría sobre la lista previamente creada en la parte LHS, colocando el *estado* de todos los objetos de esta lista a "OK". En este caso hubiera dado igual colocar el operador existencial en vez del universal.

6 Proceso del conocimiento

La base de todo el proceso de conocimiento se encuentra en la *Agenda*, mecanismo por el cual se almacenan y gestionan todos los eventos que se originan en este proceso. El procedimiento que se sigue para gestionar estos eventos no se rige por un simple algoritmo lifo o fifo. Las diferencias que se pueden destacar son:

- Se gestionan prioridades en la evaluación de las hipótesis.
- Es un mecanismo dinámico en el cual se pueden modificar la lista de eventos.
- Se incorpora la noción de resolución de conflictos ante las distintas posibilidades en la inferencia de alguna hipótesis.

La Agenda se configura entorno al concepto *centro de atención* (focus of attention), el cual no es más que la intersección de los planos de reglas y objetos. Es función de la Agenda determinar como estos planos se mueven con respecto a otros. La simplicidad de este diseño le proporciona mayor potencia y comprensibilidad, y en definitiva una mayor utilidad.

Una característica importante de la Agenda es que contiene una lista con prioridades de hipótesis, no de reglas. Esto es importante, ya que el mecanismo básico que utiliza Nexpert en el proceso de conocimiento es evaluar el estado de una hipótesis, y nunca evaluar una regla en concreto. No obstante, en su búsqueda del valor de la hipótesis, probablemente se evaluará más de una regla.

Por ello, para que Nexpert pueda empezar el proceso de razonamiento debe haber al menos alguna hipótesis en la Agenda. (Más adelante se estudiarán los mecanismos que permiten introducir hipótesis en la Agenda). Durante el proceso de conocimiento aparecerán y desaparecerán otras hipótesis nuevas en la agenda según las prioridades que tengan definidas. Así mismo, cuando ya no existen hipótesis para procesarse, la sesión finaliza.

6.1 Procesamiento de reglas

Este es el tratamiento más básico que recibe la base de conocimiento en el procesamiento de la información. Intentar la evaluación de una regla implica querer averiguar el estado de la hipótesis de la misma. El proceso que se sigue para evaluar una regla es verificar todas las condiciones de la misma según el orden en que están escritas. No obstante esta estrategia sufre variaciones, ya que se evalúan primero aquellas condiciones que tienen definidos slots con prioridad más alta.

Cuando son varias reglas las que resuelven el estado de una misma hipótesis, el estado de ésta se obtiene de la siguiente forma[2]:

- Todas las reglas han de ser *false* para que la hipótesis sea *false*.
- Al menos una regla debe ser *true* para que la hipótesis sea *true*.
- Al menos una regla debe ser *notknown* y las demás no *true* para que la hipótesis sea *notknown*.

Evidentemente, sólo se ejecutarán las acciones de aquellas reglas que han sido evaluadas como *true*. La estrategia por defecto es evaluar todas las reglas. No obstante esta estrategia se puede cambiar globalmente en la *Strategy dialog box* mediante el parámetro *Exhaustive evaluation*.

También se pueden desactivar algunas reglas con el fin de que no se evalúen ante determinados procesos de inferencia. No obstante esta facilidad se comentará cuando se estudien los mecanismos de inferencia que Nexpert provee.

6.2 Mecanismos de inferencia

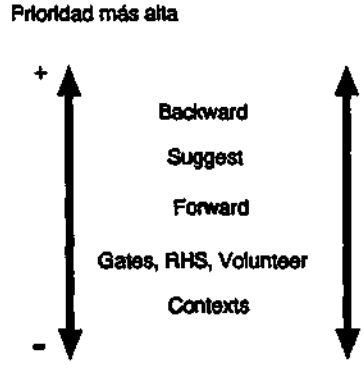
Los mecanismos de inferencia son los procesos que permiten el proceso del conocimiento. Estos producen como resultado la inclusión de hipótesis en la Agenda. Estos mecanismos permiten expandir la búsqueda de datos relevantes sin evaluar exhaustivamente

---

[2] Los estados posibles de una regla o una hipótesis son:
unknown: Aún no evaluada.
true o *false*.
notknown: Evaluada pero desconocida.

todas las reglas de la base de conocimiento.

Los siete tipos de mecanismos de inferencia que posee Nexpert pueden apreciarse en la figura 10, categorizados según las prioridades de evaluación de cada uno de ellos.



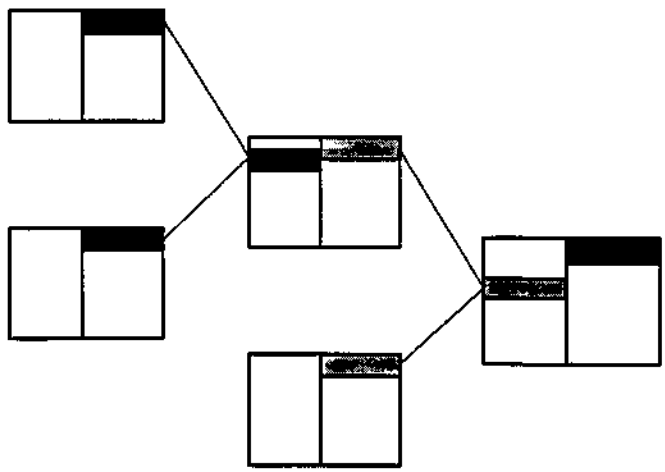
Ordenación de prioridades en la agenda
Figura 10

Antes de evaluarse cualquier hipótesis asociada a un determinado nivel, los niveles de prioridad superior no deben tener asociados hipótesis que evaluar. Localmente, dentro de un mismo nivel las estrategias funcionan según se comentó en el apartado anterior. A continuación se estudiarán cada uno de ellos.

6.2.1 Encadenamiento hacia atrás (Backward chaining).

Si en la evaluación de las condiciones de una regla, se desconoce el valor de un parámetro booleano y éste es la hipótesis de otra(s) regla(s), Nexpert lo introduce en la agenda inmediatamente para verificarlo. Estos slots, que son a la vez datos e hipótesis, se conocen como *submetas*.

El encadenamiento hacia atrás, también puede desencadenarse desde cualquier acción (RHS, OS o IC), usando el siguiente truco: *Do slot=slot*, el cual fuerza a calcular el valor de este parámetro. Los principios de encadenamiento hacia atrás se pueden aplicar recursivamente sin ninguna restricción.



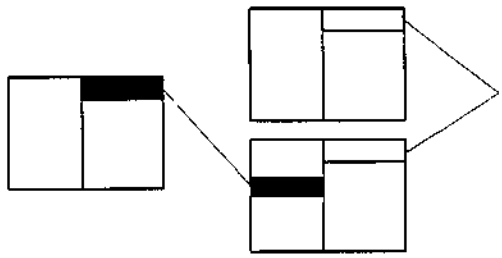
Encadenamiento hacia atrás
Figura 11

6.2.2 Sugerencia (Suggest).

Es el medio que posee el usuario para introducir hipótesis en la Agenda desde la interface de usuario. Cuando se sugiere una hipótesis, en esencia se le da a entender a Nexpert que esa hipótesis es un objetivo importante y que debe investigarse lo más pronto posible.

6.2.3 Encadenamiento hacia delante (Hypothesis forward).

Mediante este mecanismo se introducen en la agenda las hipótesis, con el objeto de ser evaluadas de aquellas reglas que tengan en su LHS una hipótesis recientemente verificada.



Encadenamiento hacia delante
Figura 12

6.2.4 Puertas (Gates).

Las Puertas *(Gates)* es el mecanismo que proporciona Nexpert para la generación automática de objetivos y el razonamiento oportunístico.

Cuando Nexpert evalua las condiciones de una regla, chequea si el slot comparado aparece en las condiciones de otras reglas. Todas las hipótesis de las reglas que contengan este slot en su parte LHS se insertan en la agenda. Esta estrategia se optimiza eliminándose aquellas hipótesis en las que se pueda deducir que la condición en la que aparece el slot es *False*.

Ejemplo.
Hipótesis evaluada *if* a.p1 > 70 *then* hipo1.h y a.p1 = 80
Hipótesis insertada *if* a.p1 > 50 *and* ... *then* hipo2.h
Hipótesis descartada *if* a.p1 > 90 *and* ... *then* hipo3.h

6.2.5 Acciones RHS

Aquellos slots que se modifican en la parte RHS de una regla y aparecen en la parte LHS de otras reglas, desencadenan un mecanismo similar a la propagación forward. Las hipótesis de estas otras reglas se introducen automáticamente en la Agenda.

Hay una diferencia importante entre los mecanismos de acciones RHS y las Puertas. Las Puertas pre-evalúan la condicion en la que aparezca el slot, y sólo introducen en la Agenda aquellas hipótesis cuyas condiciones se hayan verificado. En este mecanismo de Acciones no se preseleccionan las hipótesis.

Esta es una diferencia bastante importante, ya que en algunos casos saber que una hipótesis es falsa proporciona también bastante información.

6.2.6 Volunteer

Esta es una función que puede utilizar el usuario para interaccionar indirectamente con la Agenda de Nexpert. El usuario con este mecanismo sugiere los valores de algunos slots, insertándose automáticamente todas las hipótesis de aquellas reglas en las que aparezca en su LHS ese slot.

La filosofía del mecanismo de sugerir slots consiste en evaluar aquellas hipótesis de reglas de las que se conoce algún dato de las condiciones. Este mecanismo tiene dos propiedades muy importantes:

- No hay posibilidad de deshabilitarlo.
- Si el slot fuera usado en la condición de alguna regla mediante un operador de pattern-matching, las hipótesis asociadas a esas reglas son introducidas en la Agenda para evaluar las condiciones.

6.2.7 Contextos (Contexts o weak links)

La función de este mecanismo es unir *islas de conocimiento*. En Nexpert se conoce como isla de conocimiento aquellos grupos de reglas y objetos cuyo conjunto de condiciones, acciones e hipótesis intersectan entre ellas. En otras palabras, una isla de conocimiento es un grafo conexo de reglas. Todas las reglas que pertenecen a una isla de conocimiento, se dicen que son fuertemente conexas.

Desde un punto de vista funcional, una isla de conocimiento es un grupo de reglas y objetos que están unidos en tiempo de ejecución por cualquiera de los siguientes eventos: encadenamiento hacia atrás, encadenamiento hacia delante, puertas o acciones RHS. El mecanismo para propagar el control de una isla de conocimiento a otra son los *contextos*. Estas conexiones no se crean en tiempo de ejecución, como sucedía con los otros mecanismos de inferencia. Se definen como parte de la estructura de la base de conocimiento, uniendo dos o más hipótesis.

Debido a la baja prioridad de este mecanismo, Nexpert tiene que evaluar primero todo el conocimiento dentro de una isla de conocimiento antes de pasar a otra isla.

Los contextos son un mecanismo de propagación unidireccional. Una hipótesis puede estar en el *contexto* de cualquier número de hipótesis, y recíprocamente puede tener cualquier número de hipótesis en su contexto. Incluso, se puede definir links dentro de una isla de conocimiento, aunque esto no tenga sentido (una hipótesis puede estar dentro de su propio contexto).

6.3 Estrategias de inferencia: resolución de conflictos

En algunos casos puede ser interesante imponer un orden de evaluación en las reglas que son evaluadas. El orden en que se evalúen las reglas puede ser muy importante en muchas situaciones. Por ejemplo, si no estuviera activada la evaluación exhaustiva, los mecanismos de inferencia se pararían en el momento en que se verificara una hipótesis, esto supone el que no se evalúen algunas reglas, y por tanto no se ejecuten sus acciones.

Tal y como se había visto, dentro de la Agenda hay cinco categorías de mecanismos de inferencia (ver figura 10).

Para resolver los conflictos de inferencia se siguen los siguientes criterios:

- Las hipótesis que hayan ingresado en la Agenda con una determinada prioridad, son evaluadas antes que todas aquellas hipótesis con una prioridad menor.
- Dentro de una misma categoría, los conflictos se resuelven en función del factor *rule priority*.
- Si este factor fuera el mismo, se evalúa primero aquella que posea la condición con el factor *inference priority* más alto.
- Si durante la evaluación de una hipótesis ingresa en la Agenda una hipótesis nueva con una prioridad más alta, la evaluación de ésta se suspende momentáneamente para evaluar la nueva.

Nexpert nos da también la posibilidad de desabilitar selectivamente los mecanismos de inferencia. Para ello Nexpert hace uso del campo *rule priority* que define a una regla. Si en este campo ponemos valores negativos por debajo de ciertos umbrales, la inferencia quedará deshabilitada. Estos umbrales pueden consultarse en el manual de referencia.

6.4. Inferencia no-monotónica

En muchos casos pueden existir evidencias del estado (*entiéndase valor*) de un objeto o clase basadas en experiencias pasadas, caso más probables. Puede que estas conjeturas sean incorrectas pero, no obstante se aceptan como válidas. Ahora bien, sin en el transcurso del proceso de inferencia, se detectase un error en las suposiciones previamente tomadas, automáticamente se revisarían todas las conclusiones obtenidas a partir de dicha suposición.

Este tipo de inferencia proporciona muchas ventajas en los procesos de razonamiento, ya que esto permite que Nexpert pueda:

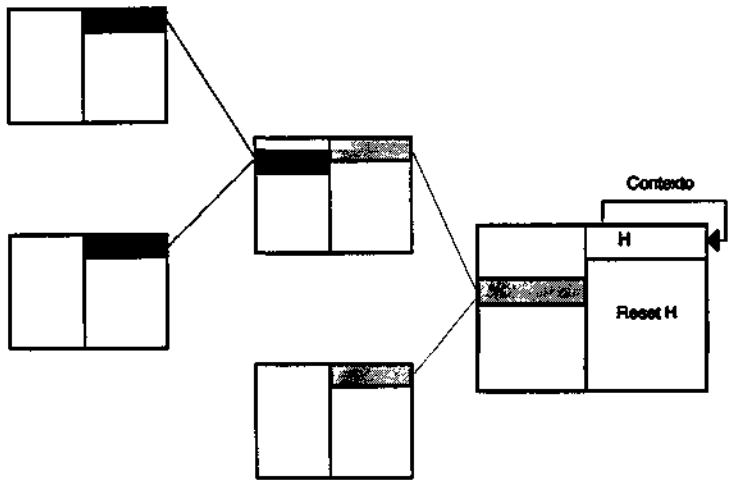
- Tomar decisiones y ejecutar acciones basándose en información incompleta.
- Revisar las conclusiones tomadas a partir de cambios en el entorno.

Existen dos mecanismos que activan este tipo de inferencia:

- **Revisiones.**
  Cuando el valor de un slot cambia (debido a una Acción RHS, Volunteer por el usuario, OS...) todas las hipótesis de las reglas que usen dicho slot en sus condiciones, entrarán en la Agenda automáticamente.
  Este tipo de inferencia se dispara automáticamente.
  Las hipótesis que aparezcan en la Agenda de esta forma, competirán con las otras hipótesis de su mismo nivel de prioridad en función de sus *Prioridades de inferencia.*

- **Reinicialización.**
  En muchas ocasiones, puede ser necesario volver a evaluar un slot, independientemente del valor que se haya obtenido previamente.
  Para estos casos, Nexpert proporciona un operador especial llamado *Reset*, el cual devuelve el estado de un slot a *unknown*. Además, si una hipótesis es reinicializada, no sólo se pondrá el estado de esa hipótesis a *unknown*, sino todas las hipótesis y condiciones de aquellas reglas que se hayan activado por el mecanismo de inferencia de encadenamiento hacia atrás. Es decir, el hecho de reinicializar el estado de una hipótesis desencadena acciones de reinicialización recursivas a través de los encadenamientos hacia atrás.
  El operador Reset puede proporcionar situaciones curiosas como la que se ilustra en la figura 13.

  En este caso, después de haberse validado la hipótesis H, se procede a efectuar una de las acciones de la parte RHS, la cual consiste en reinicializar dicha hipótesis H. Esto provocará acciones de reinicilización recursivamente a través de los encadenamientos dibujados, no sólo de las hipótesis sino también de las condiciones de las reglas. Por último, el contexto definido entre la hipótesis H y ella misma, provocará que la hipótesis H vuelva a ser introducida en la Agenda para su reevaluación. Con este truco se pueden generar los típicos bucles de la programación clásica.



Reinicialización recursiva
Figura 13

Apéndices

A. Glosario de términos

Agenda
Las hipótesis de las reglas que deben evaluarse, entran en una cola con prioridades llamada agenda. El sistema utiliza esta agenda para saber cual es la siguiente hipótesis que debe evaluar.

Clase
Funcionalmente, una clase actúa como un modelo definiendo las características que sus miembros deben poseer. Por ejemplo, todos los miembros de la clase *vehículo* poseen: precio, color y peso. Las clases pueden incluir subclases en el caso de que se necesiten otros subniveles. Por ejemplo, la clase *vehículo* podría tener otras clases como: *coches, motos, aviones...*

Hipótesis
Es la conclusión de una regla. El motor de inferencia averiguará el valor booleano de la hipótesis a través del proceso de evaluación. El resultado de este proceso convertirá el valor de la hipótesis de *UnKnown* (aún no evaluado) a alguno de los siguientes valores: *True, False o NotKnown.*

Knowcess
Proviene de **know**ledge processing. Es el comando para *arrancar* el motor de inferencia. Antes de seleccionar este comando, previamente debe haber algo en la agenda para ser evaluado.

Objeto
Los miembros de una clase son sus objetos, particularizaciones de una clase. Por ejemplo, el objeto *Mi_coche* es una particularización de la clase *coche*. Los objetos pueden heredar las características (propiedades) de las clases a las que pertenecen. A su vez, los objetos pueden poseer subobjetos, representando relaciones del tipo *es parte de*. Por ejemplo, el objeto *radio, es parte de* el objeto *Mi_coche*.

Propiedad

Las características de una clase o de un objeto son sus propiedades. Las propiedades pueden ser de uno de los seis tipos de datos que hay definidos: boolean, integer, float, string, date, time o special (para objetos que no tengan ninguna propiedad declarada).

Regla

Es la unidad más básica de conocimiento. Estructuralmente, una regla tiene tres partes bien diferenciadas. La primera parte comprende las condiciones que deben verificarse para poder obtener una conclusión. La segunda parte de la regla es la hipótesis que se verifica. Por último, la tercera parte de la regla comprende las acciones que se deben emprender tras verificarse la hipótesis.

Slot

A una propiedad particular cuando está asociada con un objeto o una clase, se le llama *slot*. El slot en la base de conocimiento es una variable que toma un valor.

Suggest

Esta es la acción específica de un usuario para introducir una hipótesis en la agenda.

Tecnología de sistemas expertos

Un tipo específico de programas que maneja el conocimiento experto de un humano. Consiste en un *motor de inferencia* y una o más *bases de conocimiento* que son creadas por el ingeniero del conocimiento para almacenar el conocimiento. Los sistemas expertos pueden necesitar o no la interacción con ususarios. Un sistema experto puede funcionar como interface con software tradicional, a modo de modulo inteligente.

Volunteer

Esta es la acción de un usuario para dar un valor a un slot de la base de conocimiento. Las hipótesis de todas aquellas reglas, en cuyo antecedente aparezca el slot al cual se le ha asignado un valor, son introducidas en la agenda para ser evaluadas.

B. Menús en Nexpert

System

Comprende todas las operaciones relativas a la configuración del sistema.

Set up environment: Establecer la configuración del sistema.

Set up printer: Seleccionar la impresora.

Save environment: Almacenar la configuración del sistema.

Quit: Fin de Nexpert Object.

Editor

Rule Editor: Editor de reglas.

Context Editor: Editor de contextos.

Object Editor: Editor de objetos.

Class Editor: Editor de clases.

Property Editor: Editor de propiedades.

Meta-slot Editor: Editor de meta-slots.

Expert

Engloba todas las operaciones de manejo de la base de conocimiento y de proceso de la misma.

Knowcess: Arrancar el proceso de conocimiento.

Restart Session: Reinicializar los valores de los slots.

Volunteer: Asignar el valor a algún slot de la base de conocimiento.

Suggest: Introducir en la agenda alguna hipótesis.

Suggest/Volunteer...:Permite hacer ambas cosas.

Strategy: Mostrar y modificar las estrategias del proceso de inferencia.

Show agenda monitor: Mostrar y modificar la agenda que debe evaluar el motor de inferencia.

Load Knowledge base: Cargar la base de conocimiento de disco a memoria.

Save Knowledge base: Grabar en disco la base de conocimiento.

Set Knowledge base: Activa una de las bases de conocimiento.

Clear Knowledge base: Borrar alguna base de conocimiento de memoria.

Report

Proporciona informes sobre el estado actual del proceso de conocimiento.

Hide transcript: Esconde una ventana con el número de licencia de la herramienta

Show current Hypot.: Durante una sesión de inferencia muestra una ventana con la hipótesis que se esté evaluando.

Show conclusions:Muestra el estado de la hipótesis que el sistema está evaluando.

Show current rule:Muestra la regla que esté ejecutando.

Case Status: Muestra todos los datos e hipótesis procesados.

Full report: Muestra todos los datos e hipótesis procesados, así como sus valores. También se puede ver la justificación de una sesión de inferencia.

Journal: Graba el estado de la sesión para posteriores usos del mismo.

Read file: Lee un fichero de datos en la memoria del sistema. Con esos datos se puede empezar el proceso de inferencia.

Write file: Graba un fichero de datos a disco.

Network

Browse Rule Network: Abre una ventana que permite ver interactivamente las reglas de la base de conocimiento.

Overview Rule Netw.: Da una visión completa de la anterior ventana, la cual es parcial.

Browse Object Netw.: Abre una ventana que permite ver interactivamente los objetos y clases de la base de conocimiento.

Overview Object N.: Da una visión completa de la anterior ventana, la cual es parcial.

Encyclopedia

List of Rules: Listado de todas las reglas.

List of Hypotheses: Listado de todas las hipótesis.

List of Data: Listado de todos los slots que aparecen en las partes LHS y RHS.

List of Objects: Listado de objetos.

List of Classes: Listado de clases.

List of Properties: Listado de propiedades.

CAPÍTULO III

ART-IM v. 2.5

Índice

1. Introducción 1
   1.1 Reglas y ejecución de los programas 1

2. Hechos 3
   2.1 Initial-Fact 3
   2.2 Definición de hechos 4
   2.3 Eliminación de hechos 5

3. Schemas 7
   3.1 Schemas y slots 7
   3.2 Herencia 9
   3.3 Definición de esquemas 11
   3.4 Definición de slots 14
   3.5 Eliminación de esquemas y slots 18

4. Reglas 21
   4.1 Ciclo de agenda 21
   4.2 Definición de reglas 23
   4.3 Declaración de salience y de grupos de reglas 23
   4.4 Condiciones 24
   4.5 Acciones 29
   4.6 Encadenamiento hacia atrás 29

5. Lenguaje de reconocimiento de patrones 33
   5.1 Patrones en la parte izquierda 33
   5.2 Patrones en las acciones de la parte derecha 39

6. Lenguaje procedural 43
   6.1 Lenguaje procedural para reglas 43
   6.2 Definiendo funciones del sistema 46

7. Variables globales 47

8. Comunicación con Excel 51

1 Introducción

ART-IM y CLIPS son dos herramientas de desarrollo de sistemas expertos cuyos fundamentos son prácticamente los mismos. Ambas utilizan el mismo algoritmo de razonamiento hacia delante: RETE. Fue la NASA la que implementó la primera versión de este algoritmo generando la primera versión de CLIPS (C Language Integrated Production System). Dada la eficiencia del mismo y el éxito que tuvo esta herramienta, a la primera versión le han seguido muchas más en los últimos años. A mediados de los ochenta Inference Corporation se aprovecha de que la NASA no posee un copyright exclusivo de la herramienta y desarrolla su propia versión del RETE tomando como referencia el código fuente de CLIPS. ART (Automated Reasoning Tool) se ha convertido en una de las herramientas que más aceptación ha tenido en el mercado, proporcionando un conjunto de paquetes de programación de propósito especial que facilitan el desarrollo y la integración del sistemas con otros lenguajes de alto nivel y paquetes como windows, bases de datos, hojas de cálculo, etc.

El presente documento intentará comentar brevemente los conceptos fundamentales y construcciones sobre las que se apoyan ART y CLIPS. Mientras no se especifique lo contrario, los comandos y construcciones funcionarán en ambos lenguajes, pero siempre dando prioridad a la sintaxis de ART.

1.1 Reglas y ejecución de los programas

Más que la típica estructura de las reglas en la forma de IF -- THEN, en ART el conocimiento heurístico toma la siguiente forma:

Cuando se verifiquen condiciones, *hacer* acciones.

Para facilitar la comprensión del funcionamiento de la herramienta, es conveniente aclarar algunas cuestiones acerca de la ejecución de los programas en ART (y CLIPS).

Normalmente las aplicaciones se cargan en memoria con el comando load. Una vez que se carga un programa, este debe resetearse con el comando *reset*. El efecto es la

inicialización de la base de datos, y la creación del conjunto inicial de hechos y esquemas definidos en la aplicación.

Se situarán en la agenda todas las reglas cuyas condiciones sean satisfechas por los datos y esquemas de la base de datos. Es posible que una regla aparezca múltiples veces en la agenda, mientras existan conjuntos de hechos y esquemas que satisfagan las condiciones.

La ejecución de un programa se realiza mediante el comando *run*. Sólo se ejecutará una regla de las que estén en ese momento en la agenda. Normalmente los cambios producidos por el disparo de la regla tendrán un efecto directo sobre el contenido de la agenda. Se añadirán unas reglas y se eliminarán otras. A partir de este momento comenzará otro ciclo de selección y disparo de una regla de la agenda.

Cuando la ejecución de un programa finaliza (no quedan reglas en la agenda), este puede reinicializarse con el comando *reset*. Si lo que se quiere es eliminar el programa de memoria, se utilizará el comando *clear*.

2 Hechos

Un hecho es un ítem de información en la base de datos. Un hecho tiene múltiples componentes, entre los que se incluyen su número, y una secuencia de uno o más elementos representando la información contenida en el hecho. Los elementos de un hecho pueden ser enteros, reales, símbolos, secuencias entre paréntesis[1], y ristras de caracteres. Un hecho típico podría ser:

f-3 (SONIDO PATO QUACK)

Cada hecho es unívocamente numerado cuando se crea. En la base pueden existir el número de hechos que sea preciso (sólo limitado por la memoria del ordenador).

Para que se pueda disparar una regla, esta tiene que encontrar algún hecho o hechos que verifiquen su antecedente. Cuando se dispara, puede eliminar hechos de la base (con el comando *retract*), y/o incluir otros nuevos (*assert*).

En general se debe utilizar una notación prefija a la hora de definir los hechos (no es obligatorio). Es decir, que el primer término del hecho debe describir una *relación* entre los términos restantes.

Ejemplo.
Para indicar que el color de parada en un semáforo es el rojo se debería utilizar un hecho de la forma:
(color semáforo rojo)

2.1 Initial-Fact

Cada vez que se resetee ART (o CLIPS) el hecho número 0 será siempre:

f-0 (INITIAL-FACT)

De esta forma tendremos un hecho inicial en la base de hechos que nos permitirá definir reglas que se disparen cada vez que se resetee y después se ejecute el programa:

[1] Sólo en ART-IM

```
(defrule ejemplo
  (INITIAL-FACT)
=>
  (printout t ' Empecemos YA'))
```

Realmente en ART el antecedente está implícito y no haría falta ponerlo.

2.2 Definición de hechos

Existen dos maneras de crear hechos:

- Pueden definirse en el fichero original utilizando el comando *deffacts*. Esto origina un conjunto de hechos que estará inicialmente presente cada vez que se resetee el programa.
- Pueden crearse en las reglas, utilizando el comando *assert*. Estos hechos no sobreviven un reset.

2.2.1 Deffacts

La sintaxis de este comando es la siguiente:

```
deffacts nombre
  [string de documentación]
  { (hecho) }*
```

- *nombre*. Es el nombre simbólico del conjunto de hechos definido con el deffacts. Esto nos permite tener más de un conjunto de hechos agrupados.
- *string de documentación*. Es una ristra opcional de caracteres que describe la información que se incluye en la sentencia.
- *hecho*. Consiste en una o más descripciones literales de hechos a ser incluidos en la base de datos cada vez que se resetee el programa.

Ejemplo.

```
(deffacts estado_inicial
(velocidad f-16 cero)
(combustible f-16 máximo)
(armamento f-16 máximo))
```

Pueden aparecer en la sentencia variables globales previamente definidas. El valor de la variable aparecerá en el hecho en el momento de ser incluido en la base de datos.

2.2.2 Assert

Se puede incluir un hecho en la base de datos desde la parte derecha de una regla o desde el bucle de comandos de alto nivel. Su sintaxis es:

(assert { hecho }+)

- *hecho*. Consiste en una secuencia ordenada de uno o más elementos. El primer elemento debe ser un símbolo. El resto puede ser cualquier objeto legal de ART o CLIPS.

Con una sentencia assert se pueden incluir uno o más hechos en la base de datos.

Ejemplo.
(assert (patos Pepe Carlos Octavio))

2.3 Eliminación de hechos

Se puede eliminar un hecho de tres formas:

- Un conjunto de hechos puede dejar de estar definido mediante el comando *undeffacts*, utilizado normalmente desde la interfase como utilidad de depuración. Estos hechos no aparecerán en el próximo reset, aunque no se eliminarán de la base de datos.
- Un hecho puede retractarse de la base de datos, ya sea a través del disparo de una regla o desde el bucle de comandos. Los hechos retractados pueden entrar en la base de datos en el próximo reset si fueron creados con una sentencia deffacts.
- Se puede borrar un hecho automáticamente de la base de datos si es dependiente lógicamente de otro dato que es eliminado.

2.3.1 Eliminación de un deffacts

Un deffacts puede dejar de estar definido utilizando el comando *undeffacts*. El conjunto de todos los hechos definidos mediante el deffacts correspondiente desaparecerá en el próximo reset; pero continuarán en la base de datos. La sintaxis es la siguiente:

(undeffacts nombre_del_deffacts)

2.3.2 Retractación de un hecho

Se puede retractar un hecho de la base de datos ya sea a través del disparo de una regla, o desde el bucle de comandos de alto nivel. Los hechos retractados podrían reaparecer en el próximo reset si inicialmente fueron creados con un deffacts. La sintaxis es:

(retract { puntero al hecho }+)

Ejemplo.

```
(defrule demuestra_retract
   ?f<- (dato 1 2 3)
 =>
   (retract ?f))
```

2.3.3 Retractación de hechos dependientes lógicamente

Si un hecho es lógicamente dependiente de otro objeto de la base de datos, y si se retracta dicho otro objeto, el hecho lógicamente dependiente también se elimina automáticamente.

Ejemplo.

```
(defrule dependencia_lógica
   (logical (h_principal))
 =>
   (assert (h_dependiente)))
(defrule retractación_automática
   (declare (salience -10))
   ?f<-(h_principal)
 =>
   (retract ?f))
```

En este ejemplo, la primera regla crea la dependencia lógica del hecho h_dependiente sobre el hecho h_principal. Cuando se dispara la segunda regla, se elimina el hecho h_principal. Este causa que también se elimine h_dependiente.

3 Schemas

En ART-IM un esquema es una colección de información acerca de un objeto particular almacenado en la base de datos. Un esquema típico podría ser el siguiente:

```
(defschema tu-perro
    (instance-of bulldog)
    (nombre Piolín)
    (color amarillo)
    (muerde muchísimo))
```

Los esquemas permiten que una aplicación considere el conjunto de datos conocidos acerca de un objeto como un todo y que actúe sobre características puntuales del mismo.

Las definiciones de los objetos de la aplicación pueden organizarse jerárquicamente, en las cuales un objeto puede heredar automáticamente determinada información incluida en las clases de datos a las que pertenece. ART mantiene la consistencia lógica de los datos que se representan con esquemas. Si durante la ejecución de un programa, se desasocia un objeto de alguna clase, ART elimina automáticamente las propiedades que se habían deducido para dicho objeto y que se apoyaban en su pertenencia a dicha clase. Esta consistencia lógica se propaga a todos los esquemas que desciendan por la jerarquía de objetos.

De igual manera, cuando se crean nuevos atributos en una clase durante la ejecución de un programa, estos se pasan automáticamente a todos los miembros de la clase y a sus descendientes.

3.1 Schemas y slots

Un schema se compone de un nombre de esquema y de cero o más slots. Cada slot representa un ítem individual de información que describe alguna característica del esquema o su relación con otros esquemas.

Un slot es una estructura de datos utilizada para representar alguna característica de un objeto. Un slot consiste de un nombre de slot, y cero o más valores. Existen dos tipos de slots:

- Los *slots atributos* que representan un atributo o una característica de un objeto (ej: nivel-de-inteligencia 65)
- Los *slots de relación* que representan una relación entre objetos (ej: is-a jefe)

Un slot se describe con un tipo especial de esquema denominado *slot schema*. Este consiste de slots definidos por el sistema utilizados para describir características del slot mismo. En definitiva existen dos tipos de esquemas:

- Esquemas normales que:
  - representan un objeto
  - heredan información
  - tienen un nombre simbólico
  - contienen cero o más slots de cualquier tipo o contenido

- Esquemas slots que:
  - describen un atributo o una relación
  - ellos mismos no heredan información, pero el uso de estos slots puede heredarse de un objeto a otro
  - tienen un nombre simbólico
  - pueden contener sólo slots específicos que indican el tipo, posibilidad de heredabilidad, y cualquier otra restricción sobre los valores del slot

Un ejemplo de esquema podría ser el siguiente:

```
(DEFSCHEMA JEFE
  (IS-A PERSONAL)
  (INGRESOS ALTO)
  (COEFICIENTE-DE -INTELIGENCIA 65)
  (TRABAJA POCO)
  (EMPLEADOS EMP1 EMP3))
```

Las jerarquías de objetos se establecen mediante enlaces jerárquicos. Algunos esquemas representan una clase de objetos, y otros representan objetos individuales. Los objetos que pertenecen a una clase comparten las características (slots) asociadas a la clase: dichos slots no tienen que duplicarse en los objetos que pertenecen a dicha clase.

3.2 Herencia

Los datos asociados a un objeto pueden pasarse, o heredarse, mediante los enlaces de herencias o relaciones. Los datos sólo pueden propagarse descendiendo por la jerarquía de objetos. Por ejemplo, si tuviéramos los siguientes objetos en la base de datos:

```
(defschema vehículo_de _transporte
  (transporta carga))

(defschema camión
  (is-a vehículo_de _transporte))
```

podríamos inferir que:

```
(defschema camión
    (transporta carga))
```

sin tener que especificarlo explícitamente.

Existen dos tipos de relaciones de herencia:

- is-a
- instance-of

Cualquier objeto conectado por un enlace is-a o instance-of a un objeto padre hereda datos del mismo. Lo contrario no se verifica. Se puede tener una cadena de dos o más enlaces del tipo is-a. Sin embargo, la heredabilidad se rompe cuando se utiliza un enlace del tipo instance-of. Este sería el último enlace de la cadena. Es decir, los datos que hereda el objeto que contiene el "instance-of" no son heredables por sus hijos. Por ejemplo, sería ilegal tener la siguiente estructura:

Ejemplo.

```
(defschema camión
  (instance-of vehículos))

(defschema mi_camión
  (is-a camión))
```

Los objetos hijos pueden sobreescribir los slots heredados. Los valores especificados en los hijos tienen preferencia sobre los heredados.

ART-IM soporta tanto herencias simples como múltiples. La herencia múltiple consiste en que un objeto posee varios padres de los que heredar valores.

Una vez que se elimina o se sustituye un valor de un slot que fue heredado por un valor específico, el valor heredado no volverá aunque se retracte el valor específico.

3.2.1 Algoritmo de herencia

Este algoritmo es válido sólo en el caso de que los slots hayan sido definidos como heredables. Si un slot es declarado como no-heredable, sus valores deben especificarse dentro de cada esquema que contengan el slot. En una sección posterior se comentará la declaración de heredabilidad.

El algoritmo difiere dependiendo de si el slot en cuestión tiene cardinalidad simple o múltiple. La cardinalidad determina si el slot puede contener uno o varios valores. Posteriormente se discutirá este aspecto.

Cardinalidad simple

El algoritmo para insertar un valor nuevo o heredado en un slot que tenga cardinalidad simple es el siguiente:

Si el nuevo valor se va a insertar en un slot vacío, se inserta el nuevo valor. Como los valores especificados tienen una prioridad mayor que los heredados, un nuevo valor especificado sobrescribe a uno heredado. Si el valor nuevo es diferente del valor actual y ambos tienen la misma prioridad, se produce un error. La descripción exacta es la siguiente:

- Si el slot no contiene ningún valor, insertar el nuevo valor.
- Si el slot contiene un valor especificado, entonces
  - Si el nuevo valor es especificado, entonces
    - Si el nuevo valor y el actual son el mismo, retener el valor actual.
    - Si los valores son diferentes, retener el valor actual y enviar un mensaje de error de cardinalidad.
  - Si el nuevo valor es heredado, retener el valor actual.
- Si el slot contiene un valor heredado, entonces
  - Si el nuevo valor es especificado, reemplazar el valor actual con el nuevo valor.
  - Si el nuevo valor es heredado, entonces
    - Si el nuevo valor y el actual son iguales, retener el valor actual.
    - Si ambos valores difieren, eliminar el valor actual y dejar el slot sin valor.

Cardinalidad múltiple

El algoritmo para slots con cardinalidad múltiple es el siguiente:

Los valores en el slot consistirán, sin duplicación, de:

- Todos los valores que se han definido como pertenecientes a este slot en este esquema - valores especificados, y
- Todos los valores que pertenecen a este mismo slot en los padres inmediatos (esquemas enlazados a este esquema por un enlace is-a o instance-of) - valores heredados.

3.3 Definición de esquemas

Un esquema normal puede entrar en la base de datos de ART de cualquiera de estas cuatro maneras:

- Utilizando la sentencia *defschema.*
- Afirmándolos mediante el comando *assert* .
- Utilizando la función *schemac*, lo que permite crear esquemas proceduralmente.
- También pueden crearse implícitamente cuando se define otro esquema. Si se define un objeto a través de un enlace de relación o de herencia a otro objeto que no se había definido previamente, se crea automáticamente el hasta entonces esquema no definido. El que un esquema creado implícitamente sobreviva un reset depende de que el esquema que causó su creación sobreviva al reset.

3.3.1 Definición de esquemas con defschema

```
defschema nombre_del_esquema
  [documentación]
  {slots}*
```

Los slots especifican un slot que describe algún atributo o una relación de *nombre_del_esquema.* Pueden haber cero o más slots en una definición de esquemas. Su forma es la siguiente:

```
(nombre_del_slot {valores}*)
```

La forma de un slot es un conjunto de uno o más elementos entre paréntesis. El primer elemento es el nombre que se desea para el slot y debe ser un símbolo. Los restantes elementos son los valores del slot.

- Si el slot es un slot de relación, el valor o valores deben ser símbolos.
- Si el slot es un slot de atributos, el valor o los valores podrán ser de cualquier objeto legal de ART.

3.3.2 Afirmando un esquema

Cuando se utiliza el comando assert para definir un esquema, se debe utilizar una sintaxis especial:

```
(assert (schema nombre_del _esquema
        {slot}*))
```

Ejemplo.

```
(defschema ventas
   (instance-of relation))

(defrule ejemplo-de- afirmación-de-esquemas
=>
   (assert (schema empleado
         (is-a personal)
         (trabaja mucho)
         (departamento ventas))))
```

Cuando se dispara esta regla, entra en la base de datos un nuevo esquema: *empleado*. Este está relacionado vía un enlace de herencia al esquema personal, tiene el atributo *(trabaja mucho)*, y está relacionado por la relación *departamento* al esquema *ventas*.

3.3.3 Creación de un esquema proceduralmente

Normalmente, los esquemas creados proceduralmente se realizan en la parte derecha de una regla. Estos esquemas pueden o no sobrevivir a un reset dependiendo del valor de su indicador de reseteo.

(schemac nombre-del-esquema &opcional indicador-de-reseteo)

- *schemac* crea un esquema vacío y lo inserta en la base de datos. Los slots serán introducidos posteriormente por otros medios (ej. con las funciones assert o slotc).
- El *indicador de reseteo* especifica si se recrea el nombre del esquema después de haber reseteado el sistema. Si este indicador tiene el valor *T* o cualquier otro que no se evalúe

a *NIL*, el esquema volverá a incluirse en la base de datos como un esquema vacío en los reseteos subsiguientes. En cualquier otro caso el esquema será eliminado de la base de datos después de un reset.

Ejemplo.

```
(defrule ejemplo
   ?f <- (nuevo-jefe ?nombre)
=>
   (retract ?f)
   (if (not (schemap ?nombre)) then
        (schemac ?nombre)))
```

Cuando se dispara esta regla, se crea un nuevo esquema si este no existía previamente en la base de datos.

3.3.4 Creación implícita de esquemas

Se pueden crear esquemas implícitamente durante el proceso de definición de otros esquemas. Si se define un esquema como relacionado a otro objeto (vía un enlace de herencia o una relación definida por el usuario), se creará automáticamente un esquema para el objeto relacionado si esta definición del esquema no existe previamente en la base de datos. Este nuevo esquema está vacío (sin slots) hasta que estos se añadan por otros medios. Existen varias posibilidades:

- Se pueden crear esquemas implícitamente por otras sentencias de defschemas.

Ejemplo.

Si tuviéramos las siguientes definiciones de esquemas:

```
(defschema ordenador
   (procesa datos)
   (requiere potencia)
   (fuente-de-potencia instalación-eléctrica))

(defschema sistema-operativo
   (instance-of relation))

(defschema micro
   (is-a ordenador)
   (sistema-operativo MS-DOS))

(defschema Mi-PC-portátil
   (instance-of  ordenadores-portátiles)
   (fuente-de-potencia baterías))
```

La base de datos contendría cinco esquemas normales: ordenador, micro, Mi-PC-portátil, MS-DOS, y ordenadores-portátiles. El esquema MS-DOS se crea implícitamente porque está relacionado vía la relación definida por el usuario al esquema micro. El esquema ordenadores-portátiles se crea implícitamente porque está relacionado vía un enlace de herencia *instance-of* al esquema Mi-PC-portátil.

- Se pueden crear esquemas implícitamente durante la afirmación de otros esquemas o slots

Ejemplo.

```
(defrule demo
=>
  (assert (schema jefe
          (is-a personal)
          (empleados EMPLEADO-3))))
```

Suponiendo que existen los esquemas personal, y empleados, la regla anterior generaría dos nuevos esquemas:

- El esquema explícitamente afirmado, JEFE
- El esquema afirmado implícitamente, EMPLEADO-3

- Por la adición de un valor a una relación de herencia o a una relación definida por el usuario. Esto es posible con las acciones assert o put-schema-value.

Un esquema que haya sido creado implícitamente sobrevivirá a un reset si el esquema responsable de su creación continúa existiendo en la base de datos después de un reseteo, y el esquema creado implícitamente es referenciado en la definición original del esquema responsable.

3.4 Definición de slots

Un slot puede entrar en la base de hechos de varias maneras:

- Se pueden definir esquemas slots, que definen características de slots, con sentencias definidas con el comando defschema. Esto crea un slot que estará presente en su estado original cuando el programa sea reseteado.
- Se pueden afirmar esquemas slots utilizando el comando assert en el consecuente de las reglas. Los slots afirmados no se reproducirán después de un reset.
- Se pueden afirmar slots en esquemas normales con la función assert en el consecuente de las reglas. No sobreviven un reset.
- Pueden crearse slots proceduralmente utilizando la función **sloc**. Sobrevivirán a un reset dependiendo del valor del indicador de reseteo.
- Los atributos de esquemas slots pueden crearse implícitamente como resultado de la definición de un esquema normal.

3.4.1 Definición de slots con defschema

```
defschema nombre-del-slot
  [documentación]
  {especificador}+
```

Un esquema de slot es un esquema especializado que describe las características de un slot particular. Su comportamiento es similar al de los esquemas normales descritos anteriormente.

El *especificador* es un slot en el esquema de slot que se utiliza para definir el comportamiento de *nombre-del-slot*. Sólo se pueden utilizar los siguientes especificadores en la definición de un esquema de slot.

```
{ (instance-of  tipo-de-slot)  |
[ (inherits  valor-de-herencia)]  |
[ (cardinality valor-de-cardinalidad)]}
```

El slot *instance-of* indica el tipo de slot; *tipo-de-slot* debe ser uno de los siguientes símbolos:

slot para un slot atributo, que representa un atributo o característica de un objeto.

relation para slot relación, que representa una relación entre objetos.

A menos que se especifique (instance-of slot) o (instance-of relation), se supone que es un esquema normal, lo que significa que los slots de cardinality y inherits no están permitidos.

El slot opcional *inherits* indica si el slot será heredable a través de los enlaces de herencia. (ej. con is-a o instance-of). El valor-de-herencia debe ser uno de los siguientes:

yes que indica que los usos de este slot son heredables vía los enlaces de herencia.

no que indica que los usos de este slot no son heredables vía los enlaces de herencia.

El slot opcional *cardinality* especifica si el slot podrá contener uno o más de un valor. El valor-de-cardinalidad debe ser uno de los siguientes:

single. El slot correspondiente sólo aceptará un valor (o también el slot vacío). Cualquier intento de asignar más de un valor generará un error.

multiple. El slot correspondiente aceptará cero o más valores.

Ejemplo.

```
(defschema empleados
   (instance-of relation)
   (inherits yes)
   (cardinality multiple))
```

Cada tipo de slot tiene características por defecto que se presuponen si no se especifican en la definición:

- Slot atributo
  - instance-of - slot
  - inherits - yes
  - cardinality - single
- Slot relación
  - instance-of - relation
  - inherits - no
  - cardinality - multiple

El tipo de slot por defecto es el slot atributo.

3.4.2 Afirmando un esquema slot

Los esquemas slots se pueden incluir en la base de datos con la función assert normalmente en la parte derecha de las reglas. La sintaxis a utilizar es la siguiente:

```
(assert  (schema nombre-de-slot
     {especificador}+))
```

Ejemplo.

```
(defrule otra-demo
=>
   (assert (schema periféricos
           (instance-of relation))
         (schema corre
           (instance-of slot)
           (inherits yes)
           (cardinality multiple))))
```

Estos esquemas no sobreviven a un reset.

3.4.3 Afirmando un slot en un esquema

Pueden afirmarse Slots y /o valores de slots en un esquema especificado que ya existía previamente en la base de datos. Esta afirmación se produce normalmente en las acciones de

las reglas utilizando la función assert. La sintaxis a utilizar es la siguiente:

```
(assert (schema nombre-del-esquema
    {slots}+))
```

slots especifica un slot describiendo nombre-del-esquema y con la siguiente sintaxis:

```
(nombre-del-slot  {valor}*)
```

En este apartado se pueden dar la siguientes circunstancias:

- Pueden afirmarse nuevos slots en un esquema ya existente. Estos slots serán heredados por cualquier esquema hijo dependiendo de su tipo y/o definición. Hacer notar que se crearán automáticamente esquemas slots para dichos slots si estos no existen previamente.
- Pueden afirmarse nuevos valores en slots existentes en un esquema.
- Si se hereda un slot de cardinalidad simple por medio de una relación de herencia, el valor afirmado sobrescribirá al heredado.
- Si el slot es de cardinalidad múltiple, simplemente se añaden los valores adicionales al conjunto de los existentes.
- Cualquier intento de afirmar un valor en un slot de cardinalidad simple que contenga un valor no heredado generará un mensaje de error.

Los slots afirmados y sus valores no sobreviven a un reset.

3.4.4 Creación procedural de slots

Pueden recrearse slots proceduralmente, a través de las acciones de la parte derecha de las reglas utilizando la función *slotc*. El valor del indicador de reseteo determinará si dicho slot sobrevivirá un reseteo.

```
(slotc   nombre-del-esquema   nombre-del-slot   indicador-de-reset)
```

Esta función crea un slot vacío en el esquema nombre-del-esquema. Si no existiera también se crearía un esquema de slot. Los valores del los slots se añadirán posteriormente por medio de las funciones assert o put-schema-value.

Ejemplo.

```
(defrule más-demos
  (schema  ?nombre
    (instance-of empleados)
= >
  (if (not (slotp ?nombre teléfono)) then
    (slotc ?nombre teléfono t)))
```

Cuando se dispara esta regla, se creará un nuevo slot para cada schema activado si dicho slot ya no existe en la base de datos. Dichos slots se reproducirán en resets subsecuentes.

3.4.5 Creación implícita de esquemas de slots.

Sólo es posible crear implícitamente esquemas de slots atributos como resultado de la definición de otro esquema normal. Los esquemas normales de relación no pueden crearse implícitamente. Cuando se define o se afirma un esquema normal, cualquier slot referenciado en la definición que no haya sido definido previamente será automáticamente creado como un esquema de slot atributo. Estos esquemas tomarán los valores por defecto de los esquemas de slot atributos. La forma usual crear estos slots es por medio de la sentencia defschema o la acción assert.

Un esquema de slot creado implícitamente sobrevivirá un reset sólo si el esquema responsable de su creación implícita está también presente en la base de datos después del reset, y además la definición del esquema responsable hace referencia a esquema de slot creado implícitamente.

3.5 Eliminación de esquemas y slots

Pueden eliminarse esquemas regulares y esquemas de slots de la base de datos de varias formas:

- **Con la sentencia Undefschema.**
  Este comando normalmente se utiliza desde el interfase como utilidad de depurado. Estos esquemas no volverán a aparecer después del próximo reset. Se eliminará de la base de datos toda la información de reseteo. Sólo permanecerá aquella información, si existiera, que entró en la base de datos por medio de alguna sentencia distinta del defschema.
  (undefschema nombre-del-esquema)

- **Retractación de esquemas y slots.**
  Pueden retractarse esquemas normales y esquemas de slots de la base de datos utilizando la función *retract*. También pueden retractarse usos de los slots con la misma función. La sintaxis a utilizar es:
  (retract (schema nombre-del-esquema
  {slot}*))

slot especifica a un slot que describe a nombre-del-esquema y tiene la sintaxis siguiente:
(nombre-del-slot {valor}*)

Cuando se especifica un slot (sin un valor) en la sentencia retract, se eliminarán del esquema especificado y de sus hijos, el slot y todos sus valores.

Si se especifica un valor concreto a eliminar, el slot no se elimina de la base de datos, sino sólo el valor especificado. Si dicho valor fue heredado por algún esquema hijo, también será eliminado de los slots especificados en dichos esquemas hijos.

El que un esquema o slot retractado reaparezca en el siguiente reset depende de como haya sido creado. Si fue creado implícita o explícitamente utilizando una sentencia defschema o la función schemac con su indicador de reseteo a verdadero, el esquema o slot reaparecerá en su forma original en el siguiente reset. En cualquier otro caso no volverá a aparecer.

- **Eliminación procedural de esquemas.**
Normalmente se utilizan las funciones schemad o expunge-schema para eliminar esquemas normales o esquemas de slots de la base de datos.

  (schemad nombre-del-esquema &opcional (resetp))
  (expunge-schema nombre-del-esquema)

No sólo se eliminarán de la base de datos el esquema especificado; también lo serán todas las referencias a dicho esquema que se establecieron por medio de slots de relación instance-of.

El argumento opcional resetp determina si eliminar la información de reseteo del esquema. Si no se especifica, la información de reseteo no se elimina, y su reaparición en el próximo reset dependerá de como fue creado. Cualquier esquema que ingrese en la base de datos implícita o explícitamente a través de un sentencia defschema o proceduralmente con el indicador de reseteo a verdadero, será recreado en el próximo reset.

Los esquemas eliminados con la función expunge-schema *no* reaparecerán en la base de datos después del siguiente reset independientemente de como hayan sido creados. (con schemad depende)

- **Eliminación procedural de usos de slots.**
  La función a utilizar en estos casos es *slotd*.
  (slotd nombre-del-esquema nombre-del-slot (&opcional resetp))

Slotd elimina el slot junto con sus valores correspondientes del esquema especificado. Los slots eliminados utilizando la función slotd pueden reaparecer en la base de datos en el próximo reset dependiendo de como fueron creados. Los slots creados implícita o explícitamente por medio de la sentencia defschema o la función slotc con el indicador de reseteo a verdadero se volverán a incluir en la base de datos en su estado original después del próximo reset. En cualquier otro caso, los slots no reaparecerán en la base de datos en el próximo reset.

4 Reglas

Una regla es un elemento de conocimiento operacional. Describe alguna acción que debería tomarse como respuesta a un determinado estado de la base de datos. La *parte izquierda de las reglas (LHS)* describe las condiciones bajo las que la regla debería actuar. La *parte derecha de la regla (RHS)* se compone de código ejecutable que ejecuta las acciones de la regla.

En teoría, una regla es una sentencia de la forma *CUANDO - ENTONCES*, como la siguiente:

CUANDO <existan estas condiciones en la base de datos>
=>
ENTONCES <ejecutar estas acciones>

Las condiciones de las reglas consisten de patrones que "reconocen" selectivamente hechos o esquemas de la base de datos. Los patrones son líneas de código que describen, con un mayor o menor detalle, los hechos que reconocerán.

Las acciones normalmente alterarán el contenido de la base de datos mediante la creación y/o la eliminación de hechos y esquemas.

Existe la posibilidad de hacer llamadas a funciones C tanto desde la parte izquierda como desde la derecha en ART-IM.

4.1 Ciclo de agenda

Las reglas interaccionan con los hechos y esquemas de la base de datos. Cuando todas las condiciones de una regla se verifican, se dice que la regla está *activada*, es decir, se ha situado en la *agenda*. Una vez en la agenda es posible, aunque no seguro, que la regla se dispare, es decir, que ejecuten las acciones de su parte derecha. Un ciclo de la agenda podría ser como sigue.

El sistema debe resetearse una vez que se ha cargado una aplicación en memoria. Esto

significa que se elimina de la base de datos cualquier hecho y esquema que pudiera contener, y se crea el conjunto inicial de esquemas y hechos declarados por la nueva aplicación.

Cuando un hecho "ingresa" en la base de datos, ya sea por causa del reseteo o porque el programa está corriendo, se compara con los patrones de varias reglas. El sistema conoce qué patrones han "reconocido" qué hechos o esquemas. Esta tarea tiene lugar en una estructura compleja de ordenación de datos denominada "red de patrones".

La salida de la red de patrones será la entrada de la red de unificación. Esta es otra estructura interna de ordenación de datos que combina la información de los reconocimientos individuales para determinar qué reglas verifican todas sus condiciones.

Si todos los patrones de una regla tienen éxito en el proceso de reconocimiento, se dice que la regla está *activada*.

La agenda es una lista de reglas activadas. Cada término de la agenda se denomina una *activación,* y representa un único conjunto de hechos y /o esquemas en la base de datos que verificaron las condiciones de una regla determinada. Es posible que la misma regla aparezca en la agenda para diferentes activaciones, dependiendo de las diferentes maneras en las que los patrones pueden ser reconocidos por los hechos y esquemas actuales. Cuando se han evaluado los hechos y esquemas, y estos han superado las redes de patrones y de unificación, la agenda está completa.

Después del reseteo, el programa está listo para correr. Entonces se disparará exactamente una *activación* de la lista de la agenda. *Disparar* una regla significa que se ejecutan las acciones de su RHS y se elimina dicha activación de la agenda.

En general no se puede predecir el orden en el que se disparará un conjunto de activaciones. Este principio de "orden aleatorio de disparo" puede modificarse con la *característica de relevancia (salience feature).* En algunas circunstancias es conveniente dar una mayor importancia a unas reglas que a otras. Es estos casos, la activación de la agenda elegirá la de mayor relevancia entre las existentes.

La ejecución de las acciones de una regla implica normalmente algún cambio sobre la base de datos, ya sea para crear nuevos hechos o esquemas o eliminar alguno ya existente. Tales modificaciones de la base generan a su vez otras alteraciones sobre la agenda como activaciones que se crean o destruyen.

Cuando los cambios de la base de datos se han evaluado en las redes de patrones y de unificación, el sistema ya está listo para disparar otra activación de la agenda. Esta por su parte alterará el contenido de la base de datos, lo que actualizará la agenda, y el ciclo continúa hasta que la agenda está vacía o el operador aborta deliberadamente el programa. Se re-evalúa la agenda cada vez que se dispara una regla.

4.2 Definición de reglas

Las reglas se definen en los ficheros de entrada y por medio de la interfase utilizando la construcción *defrule*. Una vez definido, podrían eliminarse posteriormente del sistema utilizando *undefrule*.

```
defrule nombre-de-la-regla
   [cadena de documentación]
   [declaración]
   {condiciones}*
=>
   {acciones}*
```

Las declaraciones especifican, en caso de existir, la prioridad relativa de la regla y /o el conjunto de reglas al que está asociado la misma.

4.3 Declaración de salience y de grupos de reglas

Cualquier regla escrita puede tener tanto una declaración de prioridad como de pertenencia a un conjunto de reglas. La declaración de *salience* se utiliza para establecer la prioridad relativa de una regla. La declaración de un conjunto de reglas se utiliza para agrupar reglas similares.

Una regla puede contener solamente una sentencia de declaración *(declare)*, en la que se pueden definir la prioridad y el grupo de pertenencia de la regla. La forma general de la sentencia declare es la siguiente:

(declare [(salience valor)] [(ruleset nombre)])

4.3.1 Declaraciones de prioridad

"Salience" es la prioridad relativa de la regla que se esté considerando en la agenda. Si existe más de una regla en la agenda, la regla con la mayor prioridad será la que esté en la cima de la lista y se disparará antes que las reglas de menor prioridad. Las reglas activadas con la misma prioridad se ordenarán aleatoriamente en la agenda. La sintaxis es la siguiente:

(declare (salience valor))

El valor puede ser:

- un entero dentro del rango -10.000, +10.000.
- una variable global que se evalúe a entero dentro del rango anterior
- una fórmula que se evalúe a entero dentro del rango anterior

4.3.2 Declaraciones de conjuntos de reglas

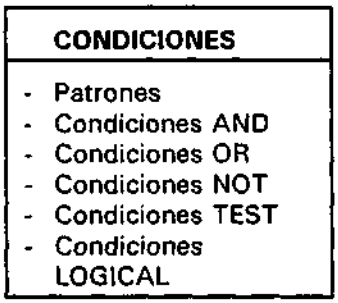
Puede que llegue a ser conveniente separar las reglas en unidades más manejables a medida que el sistema crece. Dichas unidades se denominan *conjuntos de reglas o rule sets*. Un conjunto de reglas es una colección de reglas relacionadas que están especializadas en algún aspecto de la aplicación.

Todas las reglas pertenecen al conjunto de reglas por defecto (:DEFAULT), a menos que se incluya una declaración explícita de un conjunto de reglas en la definición de una regla.

Puede evitarse que cualquier regla perteneciente a un conjunto de reglas determinado ingrese en la agenda utilizando la función *disable-ruleset*.

4.4 Condiciones

Una condición es un requisito que debe ser satisfecho antes de que una regla pueda activarse. Una regla puede tener cero o más condiciones en su parte izquierda. Existen seis tipos de condiciones:



4.4.1 Patrones

(patrón)

Un patrón sin restricciones constituye una condición simple en la parte izquierda de una regla.

Ejemplo.

```
(defrule Copa-europa-92-93
   (partido Barcelona  ?a)
=>
   (printout  t ' Gana el  '  ?a ))
```

Debe activarse el patrón (partido Barcelona ?a) por un hecho para que la regla pueda activarse.

Un patrón es una secuencia compuesta de uno o más elementos, en la que cada uno de los cuales se puede corresponder en potencia con cero o más elementos de un hecho. Los elementos pueden ser:

- Un símbolo, cadena o número; estos requieren un reconocimiento literal por los elementos de un hecho.

- Una variable a la que puede asignarse el valor del elemento correspondiente en un hecho si el resto del patrón supera también el reconocimiento. El primer carácter de una variable debe ser un ? y puede ser:
  - una variable simple (ej. ?z), la cual tomará el valor del elemento correspondiente del hecho. La asignación es local a la regla.
  - una variable libre (wildcard) (ej. ?), que puede reconocer cualquier cosa. Es realmente una marca de situación a la que no se le asignará el elemento correspondiente en el hecho.
  - una variable de segmento (ej. $?x), que puede reconocer cero o más elementos; se

asigna a la variable una secuencia de elementos reconocidos.

- una variable libre de segmento (ej. $?), que puede reconocer cero o más elementos de cualquier valor. Sólo es una marca de situación. No se le asignan valores.

Además los elementos de los patrones pueden tener:

- negaciones, utilizando ~ ; el elemento ~Barcelona FC significa reconocer cualquier hecho cuyo elemento correspondiente *no* sea el símbolo BarcelonaFC.
- disyunciones, utilizando | ; el elemento negro | azul significa reconocer o el símbolo negro o azul.
- conjunciones, utilizando & ; el elemento ?x & negro| azul significa reconocer o el símbolo negro o el símbolo azul, y asignar el símbolo reconocido a la variable ?x.
- evaluaciones, utilizando *=(fórmula)* ; el elemento =(sqrt ?z) significa reconocer cualquier número que sea igual a la raíz cuadrada del valor numérico que tiene asignado la variable ?z.
- comprobaciones, utilizando *:(test)* ; el elemento ?x&:(> ?x 100) significa reconocer cualquier número cuyo valor sea mayor que 100.

Un hecho que reconozca un patrón puede a su vez asignarse a una variable para su referencia posterior (ej. usar la función retract)

Ejemplo.
El hecho (data 3) es reconocido por:
(data ?x&:(oddp ?x) &:(< ?x 5))
y la variable ?x recibe el valor 3

4.4.2 Condiciones AND

(and {condición}+)

Una condición AND encierra una o más condiciones, las cuales pueden a su vez incluir otras condiciones and o or.

El uso más común del and en la parte izquierda de una regla es en conjunción con or, tal como muestra el ejemplo siguiente:

```
(defrule and-or
   (or (and (a)
            (b))
       (c))
 =>
   (printout t 'ala 'l Madrid' t))
```

Se producirá la activación de esta regla si *a* y *b* están presentes en la base de hechos o si lo está *c*.

Todos los patrones y test de las partes izquierdas están implícitamente encerradas en un operador and.

4.4.3 Condiciones OR

(or {condición}+)

Una condición OR encierra una o más condiciones, las cuales pueden incluir otras condiciones or.

Una condición se comporta como si se hubiera dividido la regla en múltiples reglas, de forma que cada una de las reglas contiene una de las condiciones originales.

Ejemplo.

```
(defrule ejemplo
   (or   (primero ?x)
         (segundo ?y))
=>
   (printout t 'ejemplo'))
Esta regla se comportará como si se hubieran definido estas dos:
(defrule ejemplo-uno
   (primero ?x)
=>
      (printout t 'ejemplo'))

(defrule ejemplo-dos
   (segundo ?y)
=>
   (printout t 'ejemplo'))
```

4.4.4 Condiciones NOT

(not patrón)

Una condición NOT se activará sólo cuando no existan activaciones para el *único* patrón encerrado entre paréntesis.

Las variables interiores a una condición not tienen asignaciones locales a la sentencia not. No se puede hacer referencia a las asignaciones desde fuera de la sentencia en cualquier parte de la regla. Por ejemplo, el patrón (not (foo ?x ?x)) se puede utilizar para comprobar

la no existencia de un hecho que contenga la relación foo seguida de dos elementos con el mismo valor (ej. (foo a a)). Sin embargo, al no poder imprimir un valor que no está presente, no se puede imprimir el valor de ?x o asumir que ?x utilizada en otro patrón de la misma regla contendrá el mismo valor.

4.4.5 Condición TEST

(test {fórmula})

Una condición TEST contiene código procedural, normalmente en forma de un predicado que opera sobre variables con valores asignados, que debe retornar un valor no nulo para que la regla pueda activarse.

Ejemplo.

```
(defrule ¡ejemplo!
   (primero ?x)
   (segundo ?y)
   (test (> ?x ?y))
=>
   (printout t 'ejemplo ' t))
```

Los test deben estar libres de efectos colaterales. Esto significa que una condición test debe evaluar pero nunca alterar nada en el entorno de computación. Las condiciones test se evalúan en la red de unificación, y podría ejecutarse múltiples veces en momentos impredecibles durante la ejecución del programa.

4.4.6 Condiciones lógicas

(logical {condición}+)

La condición lógica de una regla consiste en una condición que debe mantenerse activa para que los hechos afirmados por la regla continúen existiendo en la base de datos.

Las condiciones lógicas deben preceder cualquier otra condición de la parte izquierda.

Ejemplo.

```
(defrule ejemplo
   (logical (primero ?x))
   (segundo ?y)
=>
   (assert (nuevo hecho)))
```

Esta regla se disparará en presencia de los hechos:

f-3 (PRIMERO 1)
f-2 (SEGUNDO 2)

afirmando el nuevo hecho:

f-4 (NUEVO HECHO)

El (nuevo hecho) será lógicamente dependiente del hecho (primero n). Si el hecho (primero n) se elimina en algún momento, (nuevo hecho) será automáticamente eliminado en el mismo momento.

Un hecho puede recibir apoyo lógico de más de una fuente. Por ejemplo, si existieran dos hechos (primero n) presentes en la base, la regla anterior se disparará dos veces, aunque sólo se afirmará un (nuevo hecho). Este (nuevo hecho) recibirá apoyo lógico de ambos hechos (primero n). El (nuevo hecho) se eliminará de la base de datos sólo cuando se hayan eliminado todos los hechos que lo apoyan.

Pueden enlazarse dependencias lógicas en cadenas multi-generación. Es posible eliminar un único hecho y automáticamente eliminar todas las conclusiones basadas en el mismo.

4.5 Acciones

Las acciones de la parte derecha de las regla se escriben en el lenguaje procedural descrito en los manuales respectivos de las herramientas. Estas acciones modifican normalmente la base de datos, aunque también se pueden utilizar todo tipo de funciones de entrada/salida, sentencias condicionales, iteradores, y funciones definidas por el usuario.

4.6 Encadenamiento hacia atrás

Es posible simular el encadenamiento backward mediante un modelo de mensaje y réplica. El encadenamiento hacia atrás se dirige mediante un encadenamiento hacia delante a partir de *metas,* las cuales son hechos especiales. Esta meta es un mensaje situado en la base de datos que describe tan precisamente como sea posible un hecho que es necesario para poder satisfacer algún patrón especial en alguna regla de la base de conocimiento.

Si el patrón no reconoce ningún hecho conocido, se podría interpretar la nueva meta como un intento de suministrar un hecho para dicho patrón. Si el patrón reconoce alguno de los hechos de la base, la meta sería la de encontrar hechos adicionales que pueda reconocer el patrón.

Cuando se dice que la meta es *la descripción lo más precisa posible* del hecho necesario para el patrón, nos referimos a la situación en la que varios reconocimientos parciales de la regla producen diferentes asignaciones de variables. Cuando un patrón genera una meta, cualquier variable con algún valor en el patrón aporta el o los mismos a dicha meta. Si una regla tiene varios reconocimientos parciales, utilizando distintas asignaciones de variables, un patrón puede generar un gran número de metas diferentes.

Puede considerarse una meta como una *petición de ayuda* enviada por el patrón a la base de conocimiento. La meta se sitúa en la base de datos como un hecho. El patrón de la meta es el receptor del mensaje de ayuda. Normalmente es el primer patrón en una regla de encadenamiento hacia atrás. Un reconocimiento entre una meta y un patrón de meta contribuye a la activación de una o mas reglas de encadenamiento hacia atrás, y si las circunstancias son las adecuadas, alguna de ellas estará en condiciones de afirmar el hecho deseado.

Si fuera así, la regla de encadenamiento hacia atrás *respondería* al mensaje de ayuda con la afirmación de un nuevo hecho en la base de datos. La réplica es recibida cuando el nuevo hecho activa el patrón original.

En resumen, el modelo de mensajes sería:

- Una meta es un mensaje en la base de datos, el cual sirve las necesidades de un patrón concreto en una regla determinada.
- Este mensaje se recibe por un patrón de metas en una regla de encadenamiento hacia atrás.
- Si esta regla se dispara, genera una réplica en la forma de un nuevo hecho.
- Generalmente, este nuevo hecho satisfará el patrón original.

El siguiente ejemplo ilustrará el funcionamiento y la integración de este tipo de reglas.

Ejemplo.

Supongamos que se trabaja con un programa de inventario que contiene los costes de coste totales y hechos multiplicadores de venta para distintas mercaderías. Una regla backward calculará el precio de venta cuando sea necesario.

Fijemos algunos datos. Necesitaremos un multiplicador de venta y el precio de coste total de alguna mercancía.

```
f-22   (factor-multiplicador 2)
f-245  (coste camisa-11 2000)
```

El precio de venta de la camisa-11 será de 4000 ptas.

Pero, ¿cómo conseguiremos dicha información cuando la necesitemos?

Supongamos que alguien ha comprado la camisa-11 utilizando su tarjeta de crédito número tc353535. Esto añade dos nuevos hechos a la base de datos:

```
f-2055 (balance tc353535 100000)
f-3022 (cargo-en-tarjeta camisa-11 tc353535)
```

Sería fácil imaginarse una regla que añada el precio de venta de la camisa-11 a la cuenta del balance actual para obtener el nuevo balance.

```
(defrule balances
  ?x<-(cargo-en-cuenta ?item ?cuenta)
  ?y<-(balance ?cuenta ?balance)
    (precio ?item ?precio)
=>
  (retract ?x ?y)
  (assert
    (balance ?cuenta
          =(- ?balance ?precio))))
```

La tercera condición describe un precio de venta, el cual como sabemos no está presente en la base de datos. Está apunto de aparecer una meta. ¿De donde proviene esta?. Simplemente copiando la parte izquierda de la regla anterior, modificándola y generando una nueva regla:

```
(defrule meta-balance
  (cargo-en-cuenta ?item ?cuenta)
  (balance ?cuenta ?balance)
=>
  (assert (meta  precio ?item desconocido)))
```

El hecho que se afirma en esta regla es la descripción lo más específica posible del hecho necesario. El nombre del ?item es el de camisa-11, pero el ?precio es desconocido.

```
f-501 (meta  precio camisa-11 desconocido)
```

Interpretamos esta meta como un mensaje, dirigido a todas las reglas backward. Ahora hay que escribir la primera regla backward para crear los hechos de precios de venta:

```
(defrule suministra-precio-de-venta
  (meta  precio ?item desconocido)
  (factor-multiplicador ?factor)
  (coste ?item ?coste)
=>
  (assert (precio ?item
          =(* ?factor ?coste))))
```

Esta regla reconocerá la meta del hecho 501 asignando el valor camisa-11 a la variable ?item. El hecho que generará será el siguiente:

```
f-502 (precio camisa-11 4000)
```

Este nuevo hecho es la réplica a la meta. Satisface el patrón padre en la regla original

```
(defrule balances
```

```
    ?x<-(cargo-en-cuenta ?item ?cuenta)
    ?y<-(balance ?cuenta ?balance)
      (precio ?item ?precio)    ;El nuevo hecho se reconoce aquí
  =>
    (retract ?x ?y)
    (assert
      (balance ?cuenta
            =(- ?balance ?precio))))
```

El último hecho afirmado será:

```
(balance tc353535  96000)
```

5 Lenguaje de reconocimiento de patrones

Un patrón es una descripción de un dato. Se pueden utilizar en la parte izquierda de las reglas para formar relaciones entre los datos y esquemas existentes en la base y la misma regla. Los patrones en la parte izquierda especifican las condiciones bajo las que una regla podría situarse en la agenda. El lenguaje de reconocimiento de patrones permite el uso de variables, pruebas lógicas, y otras restricciones que especifican exactamente los tipos de datos que deben existir en la base de datos actual, para que una regla pueda dispararse.

También pueden utilizarse en la parte derecha de las reglas, principalmente para modificar la base de datos mediante la generación de nuevos hechos o por la creación, eliminación, o modificación de esquemas.

5.1 Patrones en la parte izquierda

Existen dos tipos de patrones utilizados en la parte izquierda de las reglas:

- patrones de hechos en lhs- reconocen hechos.
- patrones de esquemas en lhs- reconocen esquemas.[2]

5.1.1 Patrones de hechos

Describen el contenido en información de un hecho. Consiste en una secuencia ordenada de una o más restricciones:

({restricción})

Cada restricción intenta reconocer un valor o segmento de valores dentro de un hecho. Una restricción genera un test en los elementos correspondientes del hecho. Un hecho activa

[2]Sólo en ART

(instantiates) un patrón si y sólo si los contenidos del hecho satisfacen todos los test activados por las restricciones del patrón.

Las restricciones más sencillas son aquellas que son constantes, y por lo tanto, se parecen realmente a los hechos. Cada uno de estos patrones sólo reconocerán un hecho de la base.[3]

Ejemplo.
(sexo ángeles indeterminado)
(conexión nodo-1 nodo-2)
(a (b (c d e)))[4]

En general se pueden agrupar las restricciones en dos categorías:

- Restricciones simples que aplican un test a una única posición de un hecho.
- Restricciones de segmentos que aplican un test a cero o más posiciones contiguas de un hecho.

A. Restricciones simples

Especifican un test que se aplicará a una única posición dentro de un hecho. Esta posición puede ser un único término o una secuencia de términos anidados dentro del hecho.

Existen tres tipos de restricciones simples que pueden utilizarse para reconocer una única posición dentro de un hecho:

- Variable Libre o ? - comprueba si existe algún elemento en la posición correspondiente
- Restricción Conjuntiva - aplica uno o más test a una única posición dentro del hecho.
- Secuencia anidadas de restricciones - aplica test a secuencias anidadas de elementos en una posición

determinada de un hecho.

Libre.

Representada por el carácter *?*, aplica un test a una posición del hecho. El test tiene éxito si existe algún elemento en la posición correspondiente.

[3] ART no permite la duplicación de hechos en la base de datos. CLIPS posee un comando que permite explícitamente la duplicación de hechos (set-fact-duplication)

[4] Las anidación de listas sólo se permite en ART.

Ejemplo.
El patrón
(equipo liga-española ?)
reconocerá los siguientes hechos:
f-1 (equipo liga-española Barcelona)
f-2 (equipo liga-española Zaragoza)
f-3 (equipo liga-española MelillaFC)

Restricciones conjuntivas.

Estas pueden generar restricciones simples cuando aplican un test sobre un único valor dentro de un hecho como:

casa

o secuencias anidadas como:

(Mono Pato)

Pueden ser restricciones de términos o una serie de restricciones de términos conectadas lógicamente por símbolos que especifican las operaciones *and, or, not.*

Las restricciones de términos aplican un test a un único valor dentro de un hecho y puede ser de uno de estos cuatro tipos:

Restricción sobre un valor atómico.

Se corresponde con las restricciones sobre constantes como por ejemplo:

(altitud 100 metros)

Restricción sobre variables.

Actúa como una variable libre la primera vez que se encuentra en el patrón, y como una restricción sobre un valor atómico las veces subsecuentes.

Ejemplo.
El patrón
(una ?x es-una ?x es-una ?x)
reconoce el hecho
f-23 (UNA ROSA ES-UNA ROSA ES-UNA ROSA)
pero no
f-57 (UNA ROSA ES-UNA FLOR ES-UNA PLANTA)

Restricciones por fórmulas.

Consiste en la utilización de una función que devuelve un valor que es comparado literalmente con el del hecho. La restricción por fórmulas se introduce por el carácter = y entre paréntesis. Debe estar libre de efectos colaterales. Es decir, debe evaluarse siempre al mismo valor independientemente de cuando se utilice. Ej: supongamos que poseemos una

función escrita en C denominada status-c. Esta acepta como parámetro un entero que indica la edad de una persona y retorna el símbolo soltero si el argumento es menor que 30, o el símbolo casado en cualquier otro caso.

Ejemplo.
El siguiente patrón,

```
(información-personal ?persona ?edad =(status-c ?edad))
```

reconocerá

```
(INFORMACIÓN-PERSONAL "Antonio Pérez" 30 CASADO)
```

Restricción de predicados.
Se compone del carácter : seguido de una expresión entre paréntesis. La expresión debe evaluarse a NIL, o a cualquier valor distinto de NIL. Debe estar conectada con una & a una restricción de variable. Por ejemplo:

```
(tasa-de-interés ?tasa&:(< ?tasa 12))
```

Los test anteriores pueden aplicarse sobre hechos con paréntesis anidados[5]. Por ejemplo, la fórmula:

```
f-34 (a (b c) (b (c)))
```

será reconocido por:

```
(? ? ?)
(a ? ?)
(? (? ?) (b (?)))
```

etc.

Conexión de restricciones. Estas restricciones pueden conectarse lógicamente con símbolos especiales que indican la operaciones and, or, not. Estos símbolos son:

and &
or |
not ~

Con una excepción, cualquier restricción de términos puede conectarse con dichos conectores.

Excepción. Las variables deben tener un valor *antes* de que sean negadas.
Nota. La variable libre, ?, no es una restricción de término y no puede negarse o conectarse a una restricción de términos.

[5]Sólo en ART

NOT ~. Se utiliza para excluir un término en un patrón. Por ejemplo:

(color ~rojo)

se reconoce en:

f-1 (color azul)
f-2 (color (blanco azul))

AND &. Conecta lógicamente dos o más restricciones de términos o restricciones de términos negadas. Por ejemplo:

(color ~rojo&~azul) ;ni rojo ni azul

Es posible la utilización de múltiples & dentro de un patrón:

(número-par-mayor-que-10 ?x&:evenp&:(> ?x 10))

OR |. Se utiliza para conectar dos o más restricciones de términos positivas o negativas.

(vacaciones Hawaii | Rio | Tahiti | Caribe)

B. Restricciones sobre segmentos

Un segmento es una parte *secuencial* de un hecho que contiene cero o más elementos. El hecho:

f-1 (MANZANA BALÓN GATO)

contiene siete segmentos diferentes:

"vacío"
manzana
balón
gato
manzana balón
balón gato
manzana balón gato

Una restricción sobre un segmento aplica un test a un segmento para determinar si existe un ajuste entre el patrón y los hechos en la base de datos. Una restricción de segmentos puede ser de dos tipos:

- *Sobre una variable de segmento libre* $? comprueba si existen cero o más elementos en la posición correspondiente del hecho. El test siempre tendrá éxito.
- *Sobre una variable de segmento* $?nombre-de-variable aplica un test a cero o más elementos en la posición correspondiente en el hecho; si no tenía un valor anterior, comprueba si el segmento asignado se encuentra en la posición correspondiente en el hecho.

Ejemplo.
el patrón ($?x or not $x)
reconocería f-3 (TO BE OR NOT TO BE)

5.1.2 Patrones de esquemas[6]

El lenguaje de reconocimiento de patrones de esquemas es una extensión del lenguaje de reconocimiento de patrones de hechos, e incorpora el rango completo de restricciones posibles utilizadas en los patrones de hechos.

Un patrón de esquemas de parte izquierda consiste de los siguientes elementos encerrados entre paréntesis:

(schema restricción-de-esquema
{patrón-de-slot}*)

La restricción de esquema es un símbolo o un restricción simple (ej. variable, libre, fórmula) que representa el nombre del esquema que va a activar el patrón. Después, deben existir cero o más patrones de slots especificando los esquemas que podrían activar el patrón.

Ejemplo.
```
(defrule encuentra-perros-rojos
   "Presenta los nombres de los perros rojos en la base de datos excepto Fido y Fifi"
(schema ?perro & ~ Fido & ~ Fifi
   (instance-of perro)
   (color rojo))
 =>
   (printout t ?perro t))
```

Patrones de slots

Un patrón de slots consiste de uno o más elementos, encerrados entre paréntesis. Estos pueden conectarse lógicamente utilizando and, or, o not. Los elementos en un patrón de slots son:

- Restricciones de Slots: son un símbolo o cualquier restricción de patrones simple (variable, libre, ...) especificando el nombre de un atributo o relación que describe un esquema.
- Pueden haber cero o más *restricciones de valores de slots* en un patrón de slots. Estas pueden especificarse explícitamente o representarse por restricciones de patrones.

[6] Sólo en ART

Ejemplo.

```
(defschema tiene-frontera
   (instance-of relation))
(defschema California
   (is-a estado)
   (tiene-frontera  Oregon Nevada
                    Arizona México))
```

Este esquema será reconocido por los siguientes patrones:

- Este patrón se activará cuatro veces, una para cada frontera

```
(schema ?estado
   (is-a estado)
   (tiene-frontera ?frontera))
```

- Este patrón se activará tres veces, una para cada frontera que no sea México

```
(schema ?estado
   (is-a estado)
   (tiene-frontera México ?frontera))
```

- Este patrón se activará veinticuatro veces (4!), una para cada combinación de las cuatro fronteras

```
(schema ?estado
   (is-a estado)
   (tiene-frontera ?a ?b ?c ?d))
```

Reconocimiento de patrones y slots de cardinalidad múltiple

ART-IM mantiene una ordenación canónica de los valores de los slots, de forma que el orden en el que se añaden valores a un slots de cardinalidad múltiple no interfiera con las activaciones de los patrones. Es decir, una secuencia de valores de slots de dos esquemas diferentes serán iguales si contienen los mismos elementos. Esto libera al programador de preocuparse del orden de los valores de un slot cuando escribe un patrón para un slot de cardinalidad múltiple.

5.2 Patrones en las acciones de la parte derecha

De igual forma, se pueden utilizar patrones en la parte derecha de las reglas para describir datos que van a crearse, borrarse, o modificarse. Dichos patrones no invocan al reconocedor de patrones, sin que los patrones de la parte derecha pueden incorporar elementos que fueron activados en los patrones de la parte izquierda.

Un patrón de parte derecha se utiliza para formar una construcción de una acción en la parte derecha de una regla. El patrón describe el dato que va a crearse, borrarse, o modificarse. Los patrones en las acciones de las reglas son similares a los patrones de la parte izquierda, aunque no utilizan variables libre y conectivos de restricciones. Todas las variables en la parte derecha deben tener un valor o fijarlo mediante la función *bind*.

Existen dos tipos de patrones en las acciones de parte derecha:

- Patrón de hecho, utilizado para describir un dato del tipo hecho.

Ejemplo.

```
(defrule suma
   (primer-número ?x)
   (segundo-número ?y)
=>
   (assert (suma ejemplo (?x más ?y es igual a =(+ ?x ?y)))))
```

Si la base de datos contiene hechos que hacen que ?x e ?y reciban los valores 3 y 4 respectivamente, entonces esta regla creará el hecho
(suma ejemplo (3 más 4 es igual a 7))[7]

- Patrón de esquema utilizado para describir un dato esquema.

Además, un patrón de hecho puede especificarse mediante una variable que contiene un patrón de hecho de la parte izquierda de la regla, utilizando la siguiente sintaxis:

variable-local <- patrón lhs de hecho

Realmente la variable es un puntero al hecho reconocido, y puede utilizarse como argumento en los procedimientos y acciones de la parte derecha.

Ejemplo.

El patrón de la parte izquierda
?f <- (cantantes Pantoja Rocío-Jurado María-del-Monte)
reconoce el hecho:
f-9 (CANTANTES PANTOJA ROCÍO-JURADO MARÍA-DEL-MONTE)

y asigna a la variable ?f la dirección del hecho. Este puede utilizarse en la parte derecha en acciones como:
(retract ?f)

Ejemplo.

```
(defschema esposa
   (instance-of relation))
(defschema hijos
   (instance-of relation))
```

[7] La anidación de paréntesis en los hechos sólo se permite en ART-IM

```
   (defschema hijos
      (instance-of relation))
   (defschema perro
      (instance-of relation))

(defrule familia
   (Esposa-Mr_Adams ?esposa)
   (Hijos-Mr_Adams $?hijos)
   (Hijas-Mr_Adams $?hijas)
   (Perro-Mr_Adams $?perro)
=>
   (afirmar (schema familia-de-Mr_Adams
         (esposa ?esposa)
         (hijos $?hijos)
         (hijas $?hijas)
         (perro $?perro))))
```

6 Lenguaje procedural

El lenguaje procedural proporciona las principales características de cualquier lenguaje de programación procedural, incluyendo una amplia gama de funciones.

6.1 Lenguaje procedural para reglas

El lenguaje procedural se utiliza principalmente en la parte derecha de las reglas para definir las acciones de las mismas. En algunos casos también es necesario incluir código procedural en la parte izquierda de las reglas para poder definir adecuadamente las condiciones que deben cumplirse antes de que las reglas puedan situarse en la agenda. A continuación se discuten algunas de las características del lenguaje procedural.

6.1.1 Asignaciones

Las asignaciones se realizan por medio de la función *bind*. Se puede utilizar para asignar valores a variables locales o para modificar valores en las variables globales. En el caso de variables locales, estas no existirán fuera de la regla en la que se realiza la asignación.

Ejemplo.

```
(bind ?variable (- ?var 1))    ; decremento de una variable entera
```

6.1.2 Sentencia condicional

Toma la forma:

if test then {fórmula}* [else {fórmula}]

El test debe ser una condición simple.

Ejemplo.

```
=>
  (if (> edad 30) then
```

```
    (printout t 'Exento del servicio militar o social sustitutorio' t)
  else
    (printout t 'Sufrir el servicio militar o el social sustitutorio' t))
```

6.1.3 Sentencias de iteración

- **Sentencia while**

  While test do {fórmula}*

Ejemplo.

```
(defrule iteración
=>
  (bind ?var 5)
  (while (> = ?var 1) do
    (printout t '?var = ' ?var t)
    (bind ?var (- ?var 1))))
```

- **Sentencia for**

  for variable de iteración {argumentos}* do | collect$ {fórmula}*

  Para un iterador numérico:

  for variable from comienzo { to | downto} límite [by incremento] do | collect$ {fórmula}*

  Cuando se selecciona *do*, se retorna el símbolo T al final de la iteración. Si se utiliza *collect$*, se retorna una secuencia de valores coleccionados.

  La *variable* es local a la sentencia. Abandonará cualquiera de los valores recibidos durante la iteración una vez que se salga.

  Iterador es cualquier iterador legal definido en los respectivos manuales de referencia. Cuando se trate de una iteración numérica se utilizará el iterador **from**:

  Comienzo, límite, e incremento son números o cualquier expresión que se evalúe a un número.

  Es obligatorio especificar el símbolo do o collect$. do separa el iterador y cualquier argumento del código procedural. Cuando se especifica collect$, se reúne el valor de la última fórmula que sigue a collect$ para cada iteración. Al final de la iteración el for devolverá una secuencia compuesta por los valores reunidos de la última fórmula.

Ejemplo.

```
(defschema mi-pc
   (instance-of IBM-PC)
   (installed-software ART-IM CBR juegos)
   (memoria  8Mb))

(defschema Pepe-pc
   (instance-of IBM-PC)
   (installed-software Excelerator juegos)
   (memoria  12Mb))

(defschema María-pc
   (instance-of IBM-PC)
   (installed-software ART-IM juegos)
   (memoria  4Mb))

(defrule demos
   (schema ?schema
      (instance-of IBM-PC)
=>
   (printf t "< %S" ?schema)
   (for ?slot in-slots-of ?schema do
      (printf t " [ %S -" ?slot)
      (for ?value in-slot-value-of ?schema ?slot do
         (printf  t "%a " ?value))
      (printf t " ]"))
   (printf t " >\n"))
```

Esta regla se disparará una vez para cada esquema que sea una instance-of IBM-PC. Se imprimirán, para cada uno de estos esquemas, el nombre del esquema, el nombre de cada slot y todos los valores de los slots. En el ejemplo anterior la salida sería

```
< MARÍA-PC [INSTALLED-SOFTWARE - ART-IM JUEGOS] [MEMORIA 4MB] [INSTANCE-OF -
IBM-PC] >
< PEPE-PC [INSTALLED-SOFTWARE - EXCELERATOR JUEGOS] [MEMORIA 12MB]
[INSTANCE-OF - IBM-PC] >
< MI-PC [INSTALLED-SOFTWARE - ART-IM CBR JUEGOS] [MEMORIA 8MB] [INSTANCE-OF -
IBM-PC] >
```

6.1.4 Sentencia de secuenciamiento

El comando *progn* facilita la ejecución secuencial de múltiples sentencias de comandos procedurales. La parte derecha de las reglas contienen un progn implícito que abarca todas las sentencias de la parte derecha.

Su utilidad se reduce a casos como el siguiente:

```
(defrule ¿más-demo?
   ?f <- (aceptar-número)
=>
   (retract ?f)
   (if (progn
      (printout t 'Introduzca un número ')
      (bind ?num (read))
      (numberp ?num))
```

```
then (assert (data ?num))
else (printout t ?num 'no es un número' t)
  (assert (aceptar-número))))
```

6.1.5 Funciones

Es posible realizar llamadas a funciones desde la parte derecha de las reglas como si fueran cualquier otra sentencia procedural. Asimismo, existen librerías de funciones que facilitan la entrada/salida, las operaciones lógicas, cálculos matemáticos, y operaciones sobre hechos y esquemas. También se permiten llamadas a lenguajes externos.

6.2 Definiendo funciones del sistema[8]

Se pueden crear funciones definidas por el usuario utilizando *def-art-fun o def-user-fun.* def-user-fun realiza una llamada a un lenguaje externo (ver manual). Con def-art-fun se permite al usuario codificar una serie de comandos del sistema en una nueva función ART-IM que se llamará igual que cualquier otra función del sistema. Sin embargo, la nueva función def-art-fun es interpretada, en lugar de compilada. Esto ralentiza la ejecución de la misma.

```
def-art-fun nombre ({arg}*) {fórmula}*
```

arg es cero o más variables que reflejan los argumentos necesarios de la función. Todos los argumentos son obligatorios. La función retornará el valor devuelto por la última fórmula. Si no hubieran fórmulas devolvería NIL.

[8] Sólo en ART

7 Variables globales

Estas herramientas permiten un uso limitado de las variables globales mediante la sentencia *defglobal*. La sintaxis es la siguiente:

```
defglobal variable = fórmula &optional :RESET
```

defglobal se utiliza para asignar un valor a una variable. Esta podrá ser posteriormente referenciada en cualquier regla o expresión (incluyendo deffacts). Cuando se utiliza en una regla, la variable global puede referenciarse en la parte izquierda, en una acción de la parte derecha o en una declaración de prioridad. Sólo se puede utilizar una definición de variable dentro de una sentencia defglobal.

variable es un símbolo que debe comenzar por un signo de interrogación (?). La variable debe estar rodeada de asteriscos (*) (ej. ?*x*).

fórmula es cualquier expresión legal del lenguaje que devuelva un valor que será asignado a la variable.

El argumento opcional :RESET determina si la variable será reinicializada o no después de un reset. Cuando se especifique :RESET, la variable global será reinicializada al valor que se devuelva en la sentencia defglobal. Si no se especifica :RESET, la variable global mantendrá su valor actual a la hora de un reset.

Cuando se realiza un clear, las variables globales definidas por el usuario se eliminan y las variables globales del sistema se inicializan a sus valores originales.

Ejemplo.

```
(defglobal ?*número* = (+ (* 3 2) 3 4))
```

- **Referenciando una variable global en una sentencia deffacts**
  Se podría utilizar una variable global previamente definida en una sentencia deffacts. El hecho que ingresaría en la base de datos contendría el valor actual de la variable global especificada.

Ejemplo.

```
(defglobal ?*x* = 5)
(deffacts hechos
   (dato  casa ?*x* 2))
```

- **Referenciando una variable global en un patrón en la parte izquierda de una regla**
  Se supone que todas las variables en los patrones de la parte izquierda de una regla son variables locales, aunque la variable haya sido definida como global. Por lo tanto, de forma que se pueda hacer referencia al valor de una variable global en un patrón de lhs para restringir el reconocimiento de patrones, se debe insertar un símbolo = antes de la variable global.[9] El uso del signo igual hace que se evalúe la variable.

Ejemplo

```
(defglobal ?*y* = 100)

(defrule demo_local
   (data ?*y*)
 =>
   (printout t 'Variable = ' ?*y* t))

(defrule demo_global
   (data =?*y*)
 =>
   (printout t 'Variable global = ' ?*y* t))
```

La primera regla se disparará una vez para cada hecho compuesto de dos elementos, sin importar el valor del segundo elemento. La segunda regla se disparará sólo si existe un hecho (data 100) en la base de datos.

- **Referenciando una variable global en una acción de la parte derecha de una regla**
  En este caso no es necesario utilizar ninguna notación especial. La variable puede referenciarse en cualquier acción rhs.

- **Referenciando una variable global en la declaración de prioridad de una regla**
  El valor que contenga la variable global puede referenciarse directamente o por medio de una fórmula, siendo el resultado de la misma la prioridad de la regla.

Ejemplo.

```
(defglobal  ?*prioridad* = 10)
(defrule regla
   (declare (salience (+ ?*prioridad*  200))
 =>
   (printout t 'HOLA  prioridad ' t))
```

[9] Este requisito (=) tiene validez si se está utilizando el lenguaje de reconocimiento de patrones. Cuando se utiliza el lenguaje procedural en un test se considera como en el caso de la parte derecha de la regla, en cuyo caso el símbolo = sería un error.

- **Cambiando el valor de una variable global**
  Una vez definida, puede modificarse el valor de una variable global utilizando la función bind.

8 Comunicación con Excel

ART-IM permite el intercambio de datos con distintas aplicaciones WINDOWS mediante conversaciones DDE (Dynamic Data Exchange). A continuación se expone un breve ejemplo de como sería este intercambio con el EXCEL for Windows:

Para recuperar un dato de una celda de la hoja de cálculo se utilizaría la función:

get-remote celda "excel" nombre-de-la-hoja

Ejemplo.

```
(bind ?aux (get-remote "r1c1" "Excel" "base.xls")) ; celda: fila 1 columna 1
(bind ?valor (string-to-float ?aux)) ; necesario porque sólo se retornan caracteres
```

Para asignar un valor en una celda concreta:

set-remote celda valor-string "excel" nombre-de-la-hoja

Ejemplo.

```
(set-remote "r2c32" "6872" "excel" "balance.xls")
```

CAPÍTULO IV

CBR *Express* v. 1.1

Índice

1 Introducción 1

2 Interface 3

3 Búsqueda 5
- 3.1 Descripción del Caso 6
- 3.2 Respondiendo a las cuestiones 6
- 3.3 Interpretación de los resultados 7
- 3.4 Búsqueda con Seguimiento de Clientes 7

4 Construcción de una Base de Casos 9
- 4.1 El Título del Caso 9
- 4.2 La Descripción 10
- 4.3 Selección de Cuestiones 11
- 4.4 Tipos de cuestiones 11
- 4.5 Puntuación Especial 12
- 4.6 Acciones 13

5 Arquitectura del CBR Express 15
- 5.1 ART-IM 15
- 5.2 Interface en ToolBook 15
- 5.3 Bases de datos en db_Vista 16
- 5.4 Librerías C como DLL 16

6 Algoritmos de Similaridad 17
- 6.1 Puntuación de la descripción 17
- 6.2 Puntuación de las cuestiones 17
- 6.3 Similaridad de Cadenas de Textos 18
- 6.4 Similaridad Numérica 19
- 6.5 Similaridad de Caracteres 19
- 6.6 Similaridad de Palabras 20

7 Adaptación al Castellano 21
- 7.1 Puntuación 21
- 7.2 Palabras Ignoradas 22
- 7.3 Sinónimos 23
- 7.4 Eliminación de Sufijos 23
- 7.5 Modificación de las listas de preprocesamiento de texto 23

1 Introducción

CBR Express es una herramienta de construcción de sistemas expertos que se basa en la existencia de una librería de situaciones o 'casos' resueltos. Ante la aparición de una nueva situación, el sistema se encarga de recuperar el caso o los casos que mejor la identifique de forma que el usuario pueda interpretar o ajustar la solución.

Inicialmente el operador suministra una descripción del problema en lenguaje natural. A partir de esta, el sistema se encarga de seleccionar una serie de casos que podrían ser similares al nuevo problema. Cada caso de esta lista de preseleccionados aporta una batería de preguntas a realizar al usuario de forma que las respuestas de este refuerzan o descalifican la aplicabilidad del caso asociado a la nueva situación. Cada caso posee una medida de similaridad que varía desde -100 hasta +100. La búsqueda de la mejor solución termina cuando algún o algunos casos superan un determinado umbral (p.e. 90). El orden en el que se van contestando las preguntas lo decide el usuario en función de sus preferencias, oportunidad, disponibilidad de la información, etc... Después de cada respuesta el sistema actualiza la lista de casos preseleccionados, sus medidas de aplicabilidad, y la lista de cuestiones relevantes.

Cada caso en memoria se compone de una breve descripción del caso, una descripción más detallada del mismo, una batería de cuestiones relevantes, y una serie de acciones recomendadas.

2 Interface

CBR Express posee tres modos de funcionamiento dependiendo de los diferentes tipos de usuarios:

- En modo de *búsqueda* (search mode) la interface ofrece un pantalla para realizar búsquedas en la base de casos, y opcionalmente una segunda pantalla para realizar un seguimiento de los clientes (call tracking).
- En modo de *mantenimiento* (maintenance mode) el administrador de la base de casos tiene acceso a editores adicionales; uno para definir casos, otro para definir cuestiones, y otro para especificar las acciones.
- En modo de *autor* (author mode) la propia interface se puede adaptar a las necesidades particulares de cada aplicación, disponiendo de doce pantallas adicionales, que contienen scripts, buffers, y utilidades de depuración.[1]

Cada uno de estos modos de funcionamiento pueden (y deben) protegerse mediante el uso de claves de acceso a dichos modos.

[1] La propia interface de la herramienta está desarrollada en Toolbook. Para poder acceder al modo de autor es preciso disponer de una licencia de desarrollo de dicho paquete.

3 Búsqueda

El objetivo de la búsqueda por la base de casos es el de localizar un conjunto de casos que se asemejen al caso descrito por el operador. Este proceso se compone de diversos pasos.

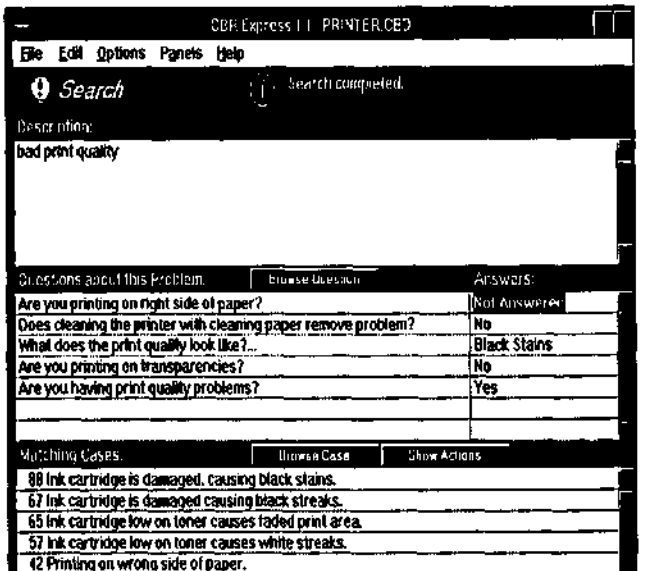


Figura 1

Primero, el operador teclea una descripción del caso actual en lenguaje natural. CBR Express realiza una búsqueda inicial de esta descripción buscando casos que posean descripciones *similares* a la misma (normalmente se muestran los cinco mejores casos). Cada caso se muestra con su medida de similaridad, un número entre -100 y 100 que muestra en qué medida se ajusta el caso la descripción de búsqueda. También se recuperan un conjunto de cuestiones para cada uno de los casos seleccionados. Estas cuestiones sirven para centrar el objetivo de la búsqueda, y para ayudar a diferenciar entre los casos competidores. Respuestas incorrectas pueden eliminar casos de la lista e incluir otros. Su efecto real consiste en modificar la medida de similaridad asociada al caso.

A medida que el usuario responde más cuestiones, el sistema actualiza la búsqueda. Las cuestiones pueden responderse en cualquier orden, y no es necesario responder a todas ellas.

El usuario continuará respondiendo cuestiones hasta que alguno de los casos tenga una medida de similaridad suficientemente alta, o hasta que todas las cuestiones hayan sido resueltas. Entonces el operador puede visualizar el caso ganador y sus acciones correspondientes.

3.1 Descripción del Caso

Esta herramienta utiliza un algoritmo de reconocimiento de texto para comparar la descripción del caso actual con las descripciones de los casos de la base de casos. Este algoritmo ignora la mayoría de las palabras usuales y se concentra en las palabras de la descripción que aportan una mayor descripción. CBR está configurado para reconocer texto en inglés (realmente americano), aunque puede modificarse para acomodarse a otro lenguaje.

Para escribir una buena descripción es necesario utilizar un lenguaje específico. Debe ser un conjunto de términos y frases comprensibles y consistentes con el vocabulario existente en la base de casos y en las descripciones de búsqueda. También es importante escribir una descripción completa; que defina la situación clara y completamente (no utilizar descripciones demasiado cortas).

3.2 Respondiendo a las cuestiones

El sistema suministrará una lista de cuestiones a responder una vez que se proporciona la descripción inicial de la situación actual. Estas cuestiones se recogen de la lista actual de casos reconocidos, y permiten diferenciar estos casos entre si. La lista de cuestiones contiene todas las preguntas asociadas con los cinco mejores casos, y ordenadas en función de la utilidad que prevea el sistema.

No es necesario responder a todas las cuestiones ni responderlas en un orden establecido. En el caso de que no se entienda alguna pregunta se dispone de una pantalla de visualización que mostrará información adicional sobre la misma. Esta información puede consistir en una descripción completa de la pregunta o una representación gráfica que ayude a responderla.[2]

[2] Diseñada con Toolbook.

Existen cuatro tipos de respuestas:

- Yes / No.
- Cuestiones que suministran una lista de respuestas legales.
- Respuestas numéricas, con limitación de rango.
- Respuestas de Texto en las que el usuario puede teclear hasta 4096 caracteres.

En cualquier caso siempre se puede dejar una respuesta como *Not Answered.*

3.3 Interpretación de los resultados

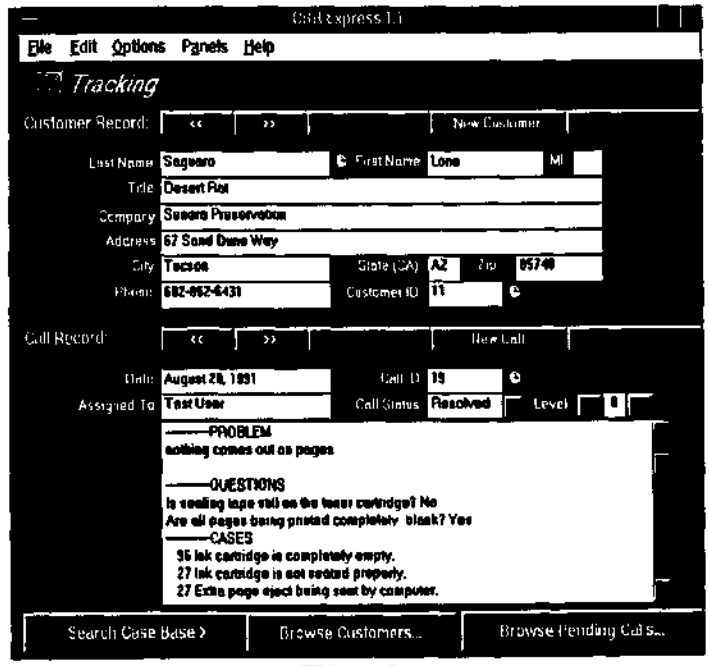
Un reconocimiento de un caso con una similaridad de 100 es posible aunque improbable debido al algoritmo de reconocimiento de casos. El umbral de aceptación dependerá del usuario y de la base de casos, de forma que la herramienta permite que el diseñador especifique su propio umbral.

Las acciones recomendadas que se especifican en cada caso de memoria pueden visualizarse a partir de la lista de casos más similares con sólo seleccionar un caso concreto. También es posible visualizar una descripción más completa del caso que puede ser en modo texto o gráfico mediante un fichero externo de ToolBook.

Cuando el sistema es incapaz de resolver un nuevo caso, es posible almacenarlo como caso no resuelto de forma que el administrador del sistema puede estudiarlo posteriormente y si fuera preciso incluirlo como un nuevo caso dentro de la librería.

3.4 Búsqueda con Seguimiento de Clientes

La herramienta facilita la integración de la base de casos con una base de datos en la que se registran las consultas y se asocian con clientes o usuarios específicos. Para ello se dispone de una pantalla en la que se detallan los datos de un nuevo cliente o se buscan los datos de uno existente. A partir de esta, se puede pasar a la pantalla de búsqueda y comenzar la consulta. Al finalizar la consulta el sistema actualizará el registro del cliente con los datos de la nueva consulta. Además el operador podría editar o añadir comentarios al registro antes de almacenarlo. Las llamadas que no puedan resolverse se almacenan en la base de datos como *llamadas pendientes* (pending calls). Esta lista de llamadas pendientes puede visualizarse, y reabrirse cuando sea necesario.



Fifura 2

4 Construcción de una Base de Casos

Para que la información que suministramos tenga alguna utilidad es necesario identificar exactamente qué es un caso. Un caso es cualquier cosa que queramos que el usuario sea capaz de encontrar. Un caso podría ser un precedente legal, algún problema de funcionamiento de cualquier tipo de maquinaria, un plan específico de batalla, etc...

Es importante remarcar que un caso debe ser importante para el usuario que realiza la consulta, porque demasiada información puede ralentizar innecesariamente la búsqueda. Siempre se debe tratar un nuevo caso como una búsqueda[3] antes de incluirlo en la base de casos. Si el sistema encuentra una alta semejanza con alguno de los existentes en la base de casos, sería preferible no incluirlo como un nuevo caso.

4.1 El Título del Caso

El título del caso es una frase de una línea que identifica unívocamente el caso. Debe definir con precisión la única característica que hace al caso importante. Esta es la información que se mostrará al usuario en la lista de casos preseleccionados durante el proceso de búsqueda. También es importante editar cuidadosamente el título para evitar errores gramaticales y de deletreo.

El título de un caso se convierte en parte de la descripción del caso a efectos de búsqueda. Esto significa que no es necesario redundar la misma información en el campo de descripción. Excepcionalmente, el título podrá ser suficiente y no será necesaria descripción alguna.

[3] El botón 'Test Case >' está para este propósito.

4.2 La Descripción

La descripción del caso consiste en un párrafo de texto que será la base para la primera búsqueda. El objeto es el de escribir una sentencia o párrafo corto que describa los síntomas del problema, o las características del caso, de la misma manera que lo haría un usuario.

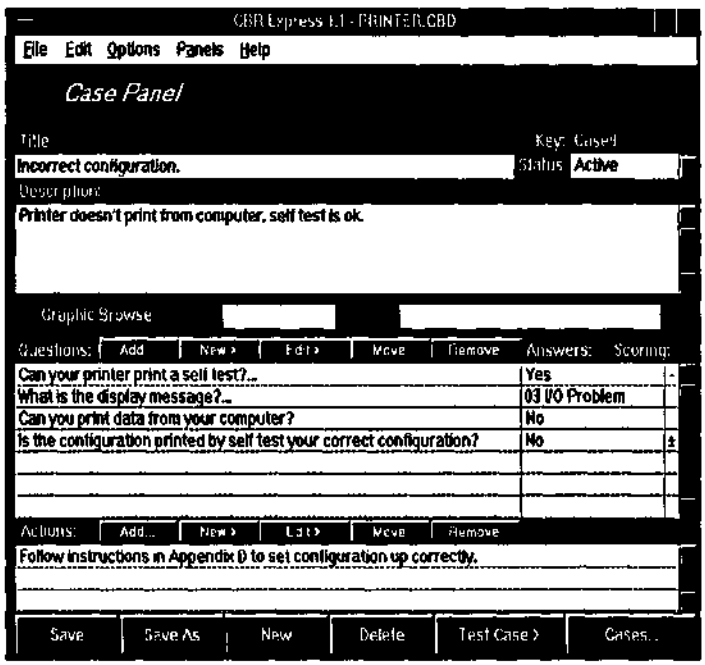


Figura 3

El sistema ignorará las principales palabras de relleno que se utilizan en las descripciones, como la mayoría de los artículos, preposiciones, verbos de ser, y sufijos[4].

Por ejemplo, si tuviéramos las siguientes descripciones

The toast always comes out too dark

Concrete hydro-electric dam anchored in basalt

se transformarían en

TOAST DARK

CONCRETE HYDRO ELECTR DAM ANCHOR BASALT

[4] La herramienta contiene una serie de *palabras a ignorar* del diccionario inglés. Para el caso del Castellano será necesario modificar dicha lista para que excluya los vocablos castellanos como artículos, preposiciones, prefijos, etc.... Posteriormente se describirá la forma en que se puede realizar esta tarea.

La única cosa que no se debe hacer cuando se escribe una descripción es sobrecargarla con términos oscuros que el usuario no será capaz de interpretar. Sería conveniente establecer un vocabulario estándar para las descripciones y entrenar a los usuarios en él (en el caso de que los usuarios no sean expertos en el dominio).

4.3 Selección de Cuestiones

Normalmente las cuestiones asociadas con un caso confirman distintos aspectos del mismo. A menudo estas cuestiones repiten y verifican la descripción del caso. También es posible utilizarlas para eliminar casos, o para confirmarlos con absoluta certeza.

Otra utilidad de las cuestiones es la de orientar el operador en un diálogo que extraiga información más precisa. Es decir, comprueban información que el usuario no incluyó en la descripción del problema.

Cada cuestión dispone de un editor que permite describir con más precisión lo que se está preguntando. En dicha pantalla se puede incluir un gráfico de ToolBook.

4.4 Tipos de cuestiones

Existen cuatro tipos de respuestas a cuestiones tal como se comentó anteriormente.

Si se define una cuestión con repuestas 'YES / NO' o una lista de respuestas legales, el sistema realiza un match de la cadena de caracteres con las de los casos de la base de casos. Si la respuesta de búsqueda coincide con la del caso, el sistema incrementa la puntuación del caso. En caso contrario, la puntuación del caso podría reducirse. La comparación no diferencia entre mayúsculas y minúsculas.

Para una respuesta de texto, CBR basa la puntuación del matching en el número de palabras que aparecen tanto en la respuesta de búsqueda como en la respuesta del caso. Si todas las palabras coinciden se concede todo el crédito. Si coinciden la mitad de las palabras, se concederá la mitad del crédito, y así sucesivamente.

En el caso de respuestas numéricas, el sistema considera el rango de valores como el 100% de amplitud de la respuesta. Si el valor de búsqueda cae dentro del 10% del rango, se

concede parte del crédito a la puntuación del matching. La puntuación es proporcional a la distancia entre los dos números. Un ajuste exacto obtendrá el crédito completo. Una falta completa reduce la puntuación del caso. Por ejemplo, si el rango de valores va desde 0 hasta 100 entonces se puede decir que el 9 es muy similar al 10. Por contra si el rango legal va de 9 a 10, los dos números están lo más lejos que pueden estarlo.

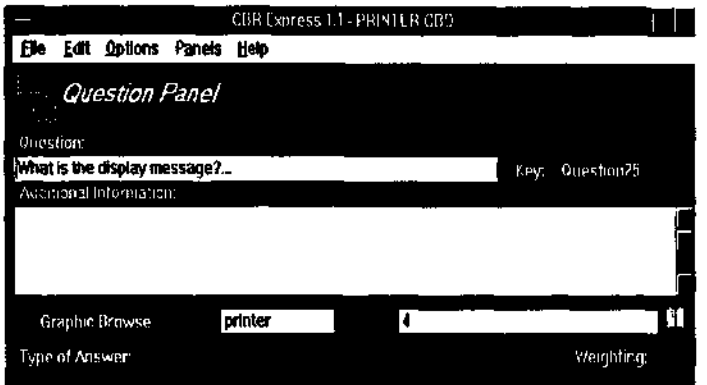


Figura 4

4.5 Puntuación Especial

Permite que el administrador asigne un comportamiento de puntuación especial a casos individuales y a la base de casos en general. En particular, el operador de búsqueda debe ser consciente de los efectos causados por los pesos de ausencia (absence weight), los de errores de emparejamiento (mismatch), y por puntuación absoluta (absolute scoring).

Cuando el operador responde una cuestión, el peso de ausencia (en el caso de estar permitido) penaliza la puntuación de todos los casos que no utilizan dicha cuestión. El propósito de esta característica es el de romper lazos entre casos que son muy similares.

El objetivo del peso por error de emparejamiento es el de penalizar la puntuación de aquellos casos que contradigan sus repuestas en la descripción de búsqueda. Son especialmente útiles en cuestiones numéricas y del tipo YES / NO, de forma que se podrá apreciar una disminución en la puntuación de aquellos casos cuyas repuestas se contradigan con la que suministra el usuario.

Las características anteriores se aplican a la base de casos como un todo. La puntuación absoluta, por el contrario se aplica a cuestiones individuales dentro de casos particulares. Significa que una respuesta correcta a *esta* pregunta en *este* caso particular confirma que al caso como el elegido. La puntuación del caso se elevará a 100. Alternativamente, se puede declarar que una respuesta incorrecta a una cuestión particular descalificaría absolutamente el caso. La puntuación del caso caerá a cero.

4.6 Acciones

Una acción podría ser una solución al problema (hacer tal cosa o sustituir la pieza XX). En un contexto más general, una acción podría considerarse como un registro de datos que asocia información adicional con el caso (por ejemplo especificar una determinada jerarquía de problemas incluidos dentro del mismo caso).

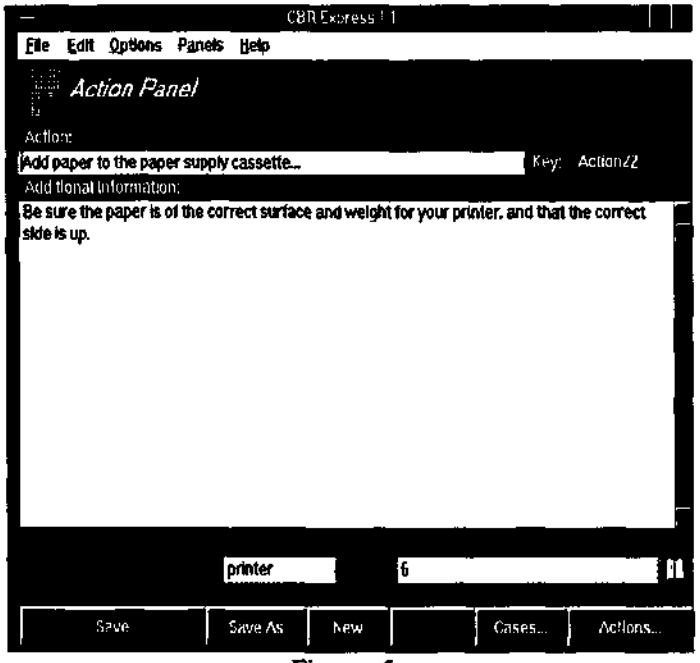


Figura 5

Distintos casos pueden compartir distintas acciones. Las acciones también pueden tener asociadas páginas gráficas de ToolBoolk.

5 Arquitectura del CBR Express

La herramienta se compone de cuatro paquetes software coordinados para generar una búsqueda dinámica de la base de casos. CBR Express posee una interface gráfica escrita en Toolbook (de Asymetric Corporation), distintas bases de datos escritas en db_Vista, y un mecanismo de reconocimiento de casos desarrollado con ART-IM (de Inference Corporation). Estos componentes principales se ejecutan sobre Microsoft Windows.

5.1 ART-IM

ART-IM es otra herramienta de construcción de sistemas expertos y es la responsable de la búsqueda en CBR. Además es posible utilizar el resto de características de esta herramienta para expandir o personalizar una aplicación.

ART-IM funciona como servidor de la interface de usuario gráfica de CBR en ToolBook. Permanece inactiva hasta que el usuario ejecuta una búsqueda. En este momento se activa y selecciona un conjunto de casos similares. Este conjunto se devuelve al ToolBook para que lo muestre al usuario.

El comportamiento por defecto del CBR es el de ejecutar ART-IM como una librería DLL transparente al usuario. Sin embargo, algunas aplicaciones requieren acceso directo al ART-IM. Esto se consigue ejecutando el ART-IM antes que el CBR. Cuando el CBR comienza a ejecutarse reconoce que ART-IM ya se está ejecutando y no invocará la versión DLL.

5.2 Interface en ToolBook

Aunque la interface de usuario escrita en ToolBook es pequeña en número de páginas (menos de veinte) contienen gran cantidad de scripts en apoyo de la función secundaria del programa: la creación de bases de casos. Esta interface gestiona las actividades de ART-IM y de db_Vista.

5.3 Bases de datos en db_Vista

db_Vista es un paquete de bases de datos que soporta registros de longitud desconocida. Esto es esencial en CBR, ya que las descripciones de los casos y las respuestas pueden tener cualquier tamaño razonable.

db_Vista funciona como un servidor tanto del interface de ToolBook como del ART-IM. La comunicación se realiza a través de las librerías de enlace dinámico de Windows (DLL).

5.4 Librerías C como DLL

Además del ART-IM, del ToolBook, y db_Vista, CBR utiliza una serie de funciones C y enlazadas al ToolBook como DLL o realmente compiladas en ART-IM. En general, estas funciones suplen la carencia de velocidad en distintos aspectos de los paquetes anteriores.

6 Algoritmos de Similaridad

En general, la tarea consiste en coger un único caso compuesto de datos del panel de búsqueda, y desarrollar una medida de similaridad con un número potencialmente alto de casos almacenados. Tanto el caso de búsqueda como los casos almacenados están caracterizados por un conjunto de características consistentes de una descripción textual junto con algunas preguntas respondidas.

6.1 Puntuación de la descripción

La puntuación de la descripción de un caso se realiza independientemente de las cuestiones del caso. Esto permite asignar a la descripción un porcentaje fijo de la puntuación total, independientemente del número de cuestiones que tenga cada caso.

Por defecto, el porcentaje destinado a la puntuación de la descripción es del 20% de la puntuación total del caso. El usuario puede modificar esta proporción si lo desea hasta el 100%.

6.2 Puntuación de las cuestiones

Las cuestiones del caso de búsqueda se puntúan individualmente contra las cuestiones correspondientes de los casos almacenados. En general, si una característica de búsqueda es igual a una característica almacenada (p.e. las dos tienen como respuesta 'Yes') la puntuación lineal del caso almacenado se incrementa por el *peso de similaridad* (match weight) de la cuestión. Este peso es un entero fijado por el usuario al definir la cuestión correspondiente. Para aquellas características que pueden resultar en una similaridad parcial (como las respuestas textuales o numéricas) la puntuación lineal del caso almacenado se incrementará en una fracción del peso definido por el usuario, dependiendo de la calidad de la similaridad. También es posible definir un peso de *error de emparejamiento* (mismatch weight) para una característica. En este caso una respuesta distinta respecto al caso de búsqueda resultará en

un decremento de la puntuación lineal del caso.

La puntuación lineal es totalizada para cada caso, y posteriormente normalizada dentro del rango asignado para la puntuación de cuestiones. La normalización restringe el valor final al rango -100 +100. Un valor normalizado de 100 indica una similaridad perfecta.

Además de los pesos anteriores para cada característica, existe una penalización por ausencia (absence penalty) aplicada sobre la puntuación lineal de cada caso almacenado. Consiste en un pequeño decremento aplicado para cada característica del caso de búsqueda que no sea compartido por el caso almacenado. Esta posibilidad está desabilitada por defecto.

Finalmente es posible designar una cuestión en un caso como poseedora de una *puntuación absoluta (absolute score)*. Se utiliza para respuestas que confirman o descalifican totalmente un caso. Esta característica está limitada a cuestiones del tipo 'Yes/No' y listas, ya que los otros tipos de cuestiones producen una puntuación parcial en la mayoría de las ocasiones.

Las secciones siguientes describirán con mayor detalle las técnicas utilizadas para calcular las contribuciones a la puntuación lineal de las cuestiones en CBR.

6.3 Similaridad de Cadenas de Textos

La similaridad de cadenas de texto (string matching) es el algoritmo utilizado por defecto para puntuar cuestiones Yes/No y listas.

Si la respuesta del caso de búsqueda (yes o no) es igual que la respuesta de caso almacenado, entonces la puntuación lineal del caso almacenado se incrementará por el peso de similaridad de la cuestión; en caso contrario la puntuación se decrementará por el peso de error de emparejamiento de la cuestión.

Si la cuestión es una que confirma absolutamente un caso, y la respuesta del caso de búsqueda es idéntica a la del caso almacenado, el peso normalizado del caso almacenado se mantendrá a 100. Si la respuesta no es idéntica el peso normalizado se fijará en 0.

Cuando existe un conflicto entre dos cuestiones absolutas, una que confirma y otra que descalifica un caso, CBR confirma el caso y se lo muestra al usuario. Técnicamente esta situación es un error.

6.4 Similaridad Numérica

Esta puntúa las respuestas numéricas en CBR. Se trata de medir la similaridad entre dos números, y asignar un incremento a la puntuación lineal (una proporción del peso de la cuestión) apoyándose en la cercanía del número de búsqueda con el número del caso almacenado. Para realizar esta comparación, es necesario conocer el rango de valores que puede tener un número. Después toma un intervalo del 10% de este rango y lo utiliza como ventana de aceptación tal como se describe a continuación:

- Si el número de búsqueda y el del caso son exactamente iguales, la puntuación lineal del caso se incrementa en el peso de similaridad de la cuestión.

- Si los dos números están alejados más del 10% del rango total, se decrementa la puntuación lineal por el peso de error de emparejamiento de la cuestión.

- Si el número de búsqueda está dentro del 10% del número del caso, se recompensa la puntuación lineal proporcionalmente a la proximidad de los dos números.

6.5 Similaridad de Caracteres

Este es el algoritmo por defecto para puntuar descripciones de búsqueda en CBR. Implica el preprocesamiento de la cadena de descripción para eliminar tanto texto ruidoso como sea posible. El sistema mantiene una lista de palabras ignoradas de forma que el preprocesador las elimina de la cadena inicial. Después elimina todos los sufijos de las palabras restantes. También se sustituyen todos los sinónimos que se hayan definido, así como los separadores entre palabras. Finalmente la cadena se pasa a mayúsculas.

En este momento el texto restante se descompone es trigramas, que son fragmentos de tres caracteres de la string. Los trigramas de la cadena de búsqueda se comparan con los de la descripción de cada caso almacenado. Se incrementará la puntuación lineal de cada caso en una fracción del peso de descripción por cada trigrama que tengan en común la descripción de búsqueda y la del caso.

6.6 Similaridad de Palabras

Este es el algoritmo por defecto para puntuar cuestiones de texto en CBR. La aproximación es similar a la anterior aunque más económica en cuestión de tiempo y memoria.

El preprocesamiento de la cadena se produce de manera similar a la aproximación anterior. Después, el texto restante se descompone en palabras en lugar de en trigramas. Se incrementará la puntuación lineal de cada caso en una fracción del peso de descripción para cada palabra que tenga en común con la respuesta de la búsqueda.

7 Adaptación al Castellano

Las funciones de puntuación, palabras ignoradas, sinónimos, y de procesamiento de texto podrían especificarse mediante la modificación del esquema CASE-BASES. La forma más apropiada de hacerlo sería a través de los mensajes del núcleo de CBR, como

```
(send add case-bases :ignored-word "de")
(send add case-bases :synonym "peter" "pedro")
```

Tal como muestra el ejemplo, el mensaje se compone de la operación a realizar (añadir o eliminar algún elemento), seguido del nombre del esquema sobre el que se va a realizar la operación (que en nuestro caso *siempre* será 'case-bases'). El tercer parámetro indica la lista de preprocesamiento de texto que se modificará en la operación. Los valores posibles son:

:separator
:ignored-word
:synonym

También se especificará/n las cadenas de texto implicadas en la operación.

El procedimiento detallado para realizar esta adaptación se comentará en el último apartado de esta sección.

7.1 Puntuación

El procesamiento de texto lo realiza normalmente el preprocesador por defecto que pasa a mayúsculas las cadenas de caracteres, elimina puntuaciones, después las palabras ignoradas, reemplaza sinónimos con palabras básicas, y finalmente elimina sufijos.

Las marcas de puntuación eliminadas por defecto son:

espacio \t \n \r ( ) - , ; = < > + / * & ? !

Los separadores no pueden aparecer en las palabras ignoradas, en los sinónimos, ni en las palabras básicas. Los separadores deben ser de longitud 1. Ejemplo:

```
(send add case-bases :separator ":")
```

Este ejemplo añade un nuevo separador a la lista anterior.

7.2 Palabras Ignoradas

La herramienta elimina determinadas palabras de las descripciones y respuestas de texto para disminuir el tiempo de procesamiento. Estas palabras son absolutamente dependientes de lenguaje y es necesario reemplazarlas para las aplicaciones en Castellano. Las palabras ignoradas inicialmente por el sistema son:

a	come	it	said	too
about	could	its	same	under
after	did	just	see	up
all	do	like	she	usually
along	each	made	should	very
also	for	many	since	was
an	from	me	some	way
and	get	might	still	we
another	got	more	such	well
any	has	most	take	were
are	had	much	than	what
as	have	must	that	when
at	he	my	the	where
be	her	never	their	which
because	herself	not	them	while
been	here	now	then	who
before	him	of	there	with
being	himself	on	these	would
between	his	only	they	you
both	how	or	this	your
but	if	other	those	
by	in	our	through	
came	into	out	thus	
can	is	over	to	

Una palabra ignorada no puede contener un separador, no puede ser un sinónimo, ni una palabra básica. Por ejemplo, si quisieramos eliminar la palabra "made" de la lista inicial de palabras ignoradas enviaríamos el siguiente mensaje:

```
(send delete case-bases :ignored-word "made")
```

7.3 Sinónimos

No existen sinónimos por defecto. La sintaxis para incluir o eliminar un sinónimo es la siguiente:

(send *operación* case-bases :synonym *palabra-básica sinónimo*)

donde *operación* debe ser add, o delete. *palabra-básica* y *sinónimo* deben ser cadenas de caracteres que no contengan separadores. Ejemplos:

```
(send add case-bases :synonym "tinta" "toner")
(send add case-bases :synonym "texto" "carta")
(send add case-bases :synonym "texto" "carácter")
```

Estos mensajes buscan sinónimos en la base de casos y los reemplazan por las palabras básicas.

7.4 Eliminación de Sufijos

La función que elimina los sufijos es específica del Inglés, y no puede modificarse por el usuario. Sin embargo, debido a que todas las funciones de procesamiento de texto son públicas, es posible escribir funciones específicas y llamarlas.

7.5 Modificación de las listas de preprocesamiento de texto

A continuación se comenta la técnica a utilizar para modificar cualquiera de las listas de puntuación, de palabras ignoradas, y de sinónimos, y adaptarlas a la aplicación que se está desarrollando.

Las listas de separadores, palabras ignoradas, y sinónimos se almacenan como slots multivaluados en el esquema CASE-BASES. Este esquema se almacena con todas las bases de casos, y se carga en ART-IM como parte de la base de casos que se abra en CBR Express. Si este esquema se modificara durante una consulta, las modificaciones también se registrarán al cerrar la base de casos .

Para realizar estas modificaciones, el primer paso es el de preparar un fichero de comandos de sinónimos, de separadores, y de palabras ignoradas de ART-IM. Llamemosle por ejemplo listas.art, y situemos dicho fichero en el mismo directorio que en el que se encuentra el fichero de interface de CBR Express. El formato para este tipo de fichero consiste en tantas sentencias 'send' como sea necesario

Estos consisten en mensajes de ART-IM enviados al esquema CASE-BASES indicándole que añada, por ejemplo, otro par de sinónimos a la lista de sinónimos. Para cada pareja, la primera palabra será reemplazada automáticamente por la segunda palabra en las cadenas de texto de CBR. Esto se realiza internamente cuando se registra un caso y durante la búsqueda, de forma que no se verán las sustituciones en la interface.

Una vez escrito el fichero de mensajes, el procedimiento es el siguiente:

1. Ejecutar la versión Windows de ART-IM
2. Ejecutar CBR Express y cargar la base de casos que necesite la modificación.
3. Ir a la ventana de comando de ART-IM y ejecutar el siguiente comando:

   (load "listas.art")

   Este cargará la lista de modificaciones. Cada línea de código se ejecutará a medida que se carga. Si existiera algún problema, seguramente sólo se conseguirá una lista parcial de modificaciones.
4. Para asegurarse de que el comando ha funcionado se puede visualizar el contenido del esquema CASE-BASES con la siguiente función desde ART-IM:

   (ppschema case-bases)
5. Si todo funciona correctamente, retornar al CBR y reindexar la base de casos Esto aplicará la tabla de sinónimos a todos los casos almacenados. Los casos que sean creados a partir de este momento ya poseerán la estructura de sinónimos definida.
6. Finalmente, volver en CBR al modo SEARCH y salir de CBR para cerrar la base de casos y hacer permanentes los cambios sobre el esquema BASE-CASES.