DESARROLLO DE UN PROTOTIPO DEL MÓDULO AGENTE PARA LA PLATAFORMA DE SIMULACIÓN GALATEA

Erasmo Gómez Molina

Trabajo presentado como requisito parcial para la obtención del grado de
INGENIERO DE SISTEMAS

UNIVERSIDAD DE LOS ANDES
MÉRIDA, VENEZUELA
Mérida, Septiembre 2002

Índice General

Índice General iii

Índice de Figuras iv

Introducción 1

1 Fundamentos teóricos 2
- 1.1 Galatea . . . 2
  - 1.1.1 Interface . . . 5
  - 1.1.2 Simulador . . . 6
  - 1.1.3 Comportamiento del simulador . . . 8
- 1.2 Agentes . . . 10
  - 1.2.1 Definición de Agentes . . . 10
  - 1.2.2 Agentes Inteligentes . . . 13
  - 1.2.3 Otros tipos de Agentes . . . 19
  - 1.2.4 Arquitectura y funcionamiento de los agentes . . . 21
  - 1.2.5 Agentes y objetos . . . 24
  - 1.2.6 Sistemas Multiagentes . . . 25
- 1.3 Representación del Conocimiento . . . 26
  - 1.3.1 Lógica Proposicional . . . 26
  - 1.3.2 Lógica Predicados . . . 27
  - 1.3.3 Reglas de Producción . . . 33
  - 1.3.4 Redes Asociativas . . . 39
  - 1.3.5 Representación mediante Plantillas . . . 40
  - 1.3.6 Representación mediante Objetos . . . 42
- 1.4 Lenguajes de programación y modelado . . . 43
  - 1.4.1 Prolog . . . 43
  - 1.4.2 JPL . . . 47
  - 1.4.3 Java . . . 47
  - 1.4.4 UML . . . 50

2 Desarrollo del prototipo **54**
2.1 Análisis y especificación de requerimientos 54
2.1.1 Determinación de los requerimientos de información 54
2.1.2 Determinación de los requerimientos de funcionales 55
2.1.3 Determinación de los requerimientos Prolog 55
2.1.4 Determinación de los requerimientos de hardware y software para la implementación del agente 56
2.2 Detalles de diseño 57
2.2.1 Galatea 57
2.3 Detalles del Agente GALATEA 58
2.3.1 Componentes 58
2.3.2 Comportamiento del modelo Agente 66
2.4 Implementación del prototipo 68
2.4.1 ConsultaThread 69
2.4.2 Dormir 71
2.4.3 Tuberia 71
2.4.4 Convertir 73
2.5 Ejemplos 75
2.5.1 Familia 75
2.5.2 Espacio en Disco 77
2.5.3 Cupido 78
2.5.4 Trenes 80
2.5.5 Cartero 82

Referencias **84**

Glosario **85**

Índice de Figuras

1 Fundamentos teóricos **2**
1.1 Esquema funcional de una federación HLA 4
1.2 Diagrama de secuencia para GALATEA 8
1.3 Estructura de un agente 11
1.4 Modelo de agente según S. Russel. y P. Norvig 12
1.5 El triángulo reactividad-racionalidad-sociabilidad 15
1.6 Diversos marcos teóricos para los agentes. Contextos de estudio (DFKI, Dr. K Fischer) 17
1.7 Agentes inteligentes según Shoham 18
1.8 Agente basado en metas 20
1.9 Agente basado en utilidad 21
1.10 Agentes y objetos 24
1.11 Representación abstracta de una plantilla. 41

2 Desarrollo del prototipo **54**
2.1 Diagrama de Caso de Uso 60
2.2 Diagrama de Secuencia 66
2.3 Diagrama de colaboración para el Agente 69
2.4 Diagrama de Clases 70
2.5 Arquitectura interna del JPL 83

Introducción

GALATEA es una plataforma de simulación que soporta extensiones al formalismo DEVS que permiten incluir la Orientación por Agentes en la simulación de Sistemas. Esta plataforma [Dávila–Uzcátegui:2000] ha sido concebida como un sistema orientado a objetos, intrínsecamente paralelo y potencialmente distribuido. Donde cada agente es simulado por un proceso relativamente independiente y el conjunto de procesos podría, eventualmente, distribuirse sobre más de una plataforma de cómputo con miras a la simulación interactiva. Más aún, un agente simulado puede representar a un individuo, a grupos sociales, o a instituciones que se comportan como un individuo, en ciertas circunstancias.

Por esta razón, nosotros construimos el módulo Agente para la plataforma de simulación GALATEA, proporcionando las herramientas necesarias para incorporar un agente inteligente a esta plataforma siguiendo los pasos de diseño del equipo de desarrollo.

La arquitectura del módulo agente construido para la plataforma GALATEA, es una estructura de alto nivel programado bajo orientación a objetos, su motor de inferencia está desarrollado en Java, para un agente donde su base de conocimiento esta programada en Prolog. Teniendo como resultado un conjunto de esquemas, subsistemas, clases, que se conectan a GALATEA y que puede tener asignaciones de responsabilidades y colaboraciones entre objetos de manera simultánea y distribuida.

Luego de haber hablado en líneas generales de lo que queremos realizar en este trabajo que servirá como proyecto de grado, hablaremos un poco de como se desarrollará cada uno de los capítulos, queremos enfocarlos de manera clara y concisa y así, cualquier lector que quiera leer este manuscrito pueda entender el trabajo realizado sin ser un experto. En el primer capítulo, hablamos de todos los fundamentos y conceptos teóricos necesarios que permitan proporcionar a la plataforma de simulación GALATEA, el soporte de agentes para la simulación de sistemas. En el segundo capítulo, se describen a través de una documentación técnica los pasos de diseño e implementación del prototipo, que permite integrar el simulador GALATEA, con el agente realizado en prolog.

Capítulo 1

Fundamentos teóricos

GALATEA, es una plataforma para la simulación de sistemas bajo los enfoques distribuidos, interactivos, continuos, discretos y combinados; desarrollado en la Universidad de los Andes, que combina el GLIDER con una familia de lenguajes lógicos específicamente diseñados para el modelado de agentes. Este capitulo esta destinado al enfoque teórico necesario para el desarrollo del prototipo del módulo Agente para GALATEA. En la sección uno, hablamos sobre el Diseño de la Plataforma de Simulación Galatea. En la sección dos, nos concentramos a estudiar los sistemas multi-agentes inteligentes: definición de agentes, tipos de agentes, arquitectura. En la sección tres hacemos un bosquejo general sobre las diferentes representaciones del conocimiento, en la sección cuatro hacemos referencia sobre los lenguajes de programación Prolog, Java, JPL y el lenguaje de Modelados UML.

1.1 Galatea

La plataforma de la simulación GALATEA integra los conceptos y herramientas que permiten simular sistemas bajo los enfoques distribuido, interactivo, continuo, discreto y combinado. Además, en esta plataforma, incorporamos soporte para el modelado y la simulación eficiente de sistemas multi-agentes.

Nuestra intensión es hacer de la plataforma una herramienta que facilite las tareas de modelado y simulación de sistemas, pensando en esto, proponemos que GALATEA este conformada por:

- una familia de lenguajes, y sus respectivos compiladores, que permiten simular sistemas multi-agentes,
- un simulador

- un ambiente para la construcción de modelos.

Esta plataforma está desarrollándose en Java [Sun Microsystems Inc: 1994] y podrá ser ejecutada desde aquellos ambientes que soportan aplicaciones Java. Hasta ahora las implementaciones asociadas a esta plataforma han sido parcialmente evaluadas en Windows y Linux:

- En [Dávila:1997] se muestra una especificación detallada de los lenguajes.
- En [Uzcátegui:2002], se muestran los detalles de diseño de la plataforma de simulación y los detalles de diseño e implementación de un prototipo funcional del simulador.
- En [Cabral:2001] se muestran los detalles de diseño e implementación del prototipo funcional para la interfaz gráfica del ambiente de desarrollo.

GALATEA propone una reformulación de la manera tradicional de manejar la relación entre los componentes de simulación con miras a la simulación interactiva, por ello es necesario pensar en varios programas ejecutándose en computadoras heterogéneas y distribuidas y que interactúan a través de un sistema operativo distribuido. Nosotros aprovechamos estas características para permitir el modelado y la simulación de sistemas multi-agentes.

Como mencionamos anteriormente, los adelantos en Simulación Interactiva han permitido el desarrollo de un marco de referencia estándar que facilita la integración de múltiples componentes de simulación. Este marco de referencia, denominado HLA (ver sección 1.1.4) en [Uzcátegui:2002], nos sirve como referencia para el diseño de la plataforma.

La estructura de la federación HLA, usada en GALATEA figura 1.1, esta dividido en diferentes componentes funcionales principales. El primer componente, la Simulación, reúne todas las simulaciones (como se define en HLA) el cual debe incorporar capacidades específicas que permiten a los objetos de una simulación, federados, conectarse con los objetos de otra simulación. El intercambio de datos entre federados es soportado por los servicios implementados en la interface. Nosotros particularizamos la simulación HLA para que cada federado mantenga su propio subconjunto del modelo del mundo e incluimos:

1. un federado que representa el simulador principal que controla todos los eventos. El uso de un único simulador puede rendir beneficios sustanciales, incluyendo:

    - facilita la validación del comportamiento de los sistemas existentes y la reutilización de los modelos existentes,

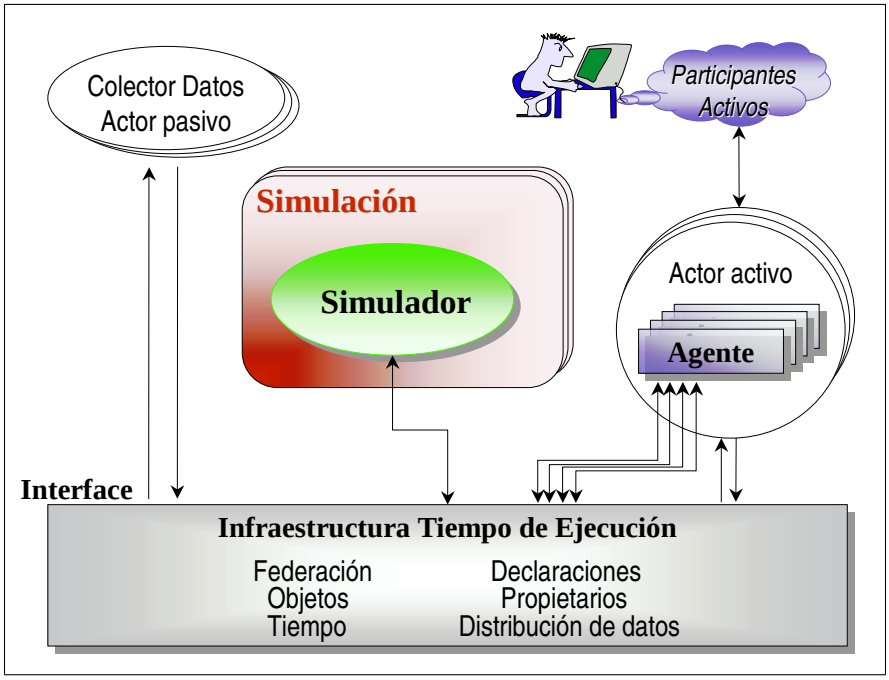

Figura 1.1: Esquema funcional de una federación HLA

- proporciona una infraestructura para desarrollar nuevos sistemas,
- facilita la comparación de resultados.

2. un grupo de federados que representa los agentes del sistema. Estos agentes poseen algún grado de comportamiento autónomo y comunicación con su ambiente y con los otros agentes para alcanzar sus metas. El comportamiento de los agentes causa cambios en el ambiente y en consecuencia el ambiente reacciona.

El segundo componente funcional de la arquitectura es la Infraestructura de Tiempo de Ejecución (*RTI: Runtime Infrastructure*). Esta infraestructura proporciona un grupo de servicios de propósito general que soportan las interacciones entre los componentes de simulación. Todas las interacciones entre los componentes de la simulación se llevan a cabo a través del RTI.

El tercer componente es la Interface. Esta interface, independiente de su aplicación, mantiene una manera estándar de relacionar los componentes de simulación con el RTI. La interface es independiente de los requisitos para el modelado y del intercambio de datos entre los componentes de la simulación. En particular, en nuestro caso, la interface además de controlar la sincronización del tiempo global, debe proporcionar el conjunto de servicios necesarios para que el intercambio de datos entre los agentes y la simulación a la que están asociados fluya.

A continuación se da una descripción sobre los diferentes componentes de la arquitectura propuesta por [Uzcátegui:2002], tal es el caso de la interface, el simulador y los agentes. Esta descripción incluye los detalles que deben tomarse en cuenta en la implementación.

1.1.1 Interface

En general en GALATEA cada una de las entidades de nuestra federación HLA se asocia a un objeto. Estos objetos interactúan a través de pase de mensajes los cuales son vistos como eventos puntuales. Como mencionamos anteriormente tanto los modelos de simulación como los agentes corresponden a federados en nuestro modelo HLA y por ende siguen el siguiente ciclo de vida:

1. Se incorpora a la federación.
2. Establece sus requerimientos de datos.
3. Revisa y encuentra las instancias de objetos que necesita.
4. Actualiza los valores de sus atributos.
5. Avanza el reloj.
6. Elimina objetos.
7. Se separa de la federación.

Los federados necesitan sincronizar sus operaciones a medida que avanza el tiempo, por tanto en cada actualización cada federado cumple repite los pasos 2-6 y cada actualización es potencialmente síncrona con respecto a un tiempo virtual compartido. Con la finalidad de reducir el tráfico de mensajes y limitar la cantidad de interrupciones en cada federado HLA propone mecanismos que permiten reducir el número de mensajes:

Publicación. Al iniciarse la ejecución de una federación cada simulación registra sus objetos y atributos en la RTI.

Inscripción. Además debe registrar que tipo de atributos externos necesita para llevar a cabo su tarea. También es posible definir un rango para los valores que puede asumir dicho atributo.

El objetivo de estos mecanismos es extraer tanta información como sea posible de las simulaciones. Tanto la inscripción como la publicación son mecanismos dinámicos y pueden ser alterados durante la ejecución de la sesión. Cabe destacar que el RTI hace distinción entre direccionamiento de datos (establece la conexión necesaria) y el envío propiamente dicho de los datos. La meta al definir el RTI es establecer la red

de conexiones necesarias para minimizar el retardo y maximizar el desempeño. Este factor es de vital importancia en el caso en que la RTI es utilizada por federados que se encuentran distribuidos en una red de computadores.

En vista de que nuestros federados intercambian atributos, necesitamos definir un mecanismo para compartirlos. Por lo pronto podemos obligar a que cada atributo sea controlado por el federado que instancia el objeto que lo contiene. No obstante, el federado que es dueño del atributo es el único que puede alterarlo durante la ejecución.

Con respecto al manejo del tiempo, de momento se ha sugerido que la sincronización del tiempo global sea realizada a través de la supervisión de actividades y la concurrencia se resuelva en el modelado. Los puntos de control de simulación se consideran eventos. Éstos puntos de control son dispositivos especiales que permiten controlar tanto las percepciones y las acciones de los agentes como los progresos de la simulación. Además, como en simulación DEVS clásica, se actualizan las variables que representan el estado del ambiente después de cada avance de tiempo.

Dado que en [Zeigler et al.: 2000] se muestra una implementación natural para DEVS haciendo uso de un marco de referencia orientado por objetos, la cual se utiliza en [Zeigler:1999] para crear simulaciones HLA, explicando como es el ambiente DEVS/HLA, como se realiza el mapeo DEVS/HLA, y además se presentan algunos detalles de la implementación en C++ del protocolo de simulación DEVS en HLA, proponemos que se sigan estos trabajos al momento de implementar la interface en lenguaje Java para nuestra plataforma.

1.1.2 Simulador

Como mencionamos anteriormente el simulador de la plataforma GALATEA es una extensión del simulador GLIDER. Por tanto, la presente descripción estará basada las modificaciones necesarias para que el simulador GLIDER presentado en [Uzcátegui:2002] pueda integrarse a la plataforma.

Algoritmo general para el simulador

La simulación de un sistema consiste en generar y activar una sucesión de eventos en el ambiente simulado. Por consiguiente, el simulador, esencialmente, activa dichos eventos y ejecuta las piezas de código asociadas con cada uno. Se puede argumentar que el disparo o activación de un evento consiste, precisamente, en la ejecución de esa pieza de código. Sin embargo, hay casos que no encajan con esta definición. Así, en GLIDER un evento es la activación de un nodo que puede o no implicar la ejecución de su código.

Una simulación de un modelo GLIDER es la activación de los eventos relacionados a los nodos y la posterior revisión de la red, buscando el código a ser ejecutado y probando sus pre-condiciones estructurales. Si estas pre-condiciones se alcanzan,

el código debe ejecutarse. Para garantizar que los efectos (cambios) inducidos en cada evento se propaguen a través de todo el sistema, la activación de un nodo (la ocurrencia de un evento identificado con el nombre del nodo) puede activar nuevos eventos y además puede implicar cambios en el estado global del sistema o pases de mensajes.

Así, una revisión de la red conduce a la evolución de la simulación del sistema. Cuando un nodo se activa, su código se ejecuta y se inicia la revisión del resto de la red. Durante la revisión, aquellos nodos que no tienen una lista externa (*EL: External List*), lista de espera para los mensajes, o cuya EL está vacía se ignoran. En estos nodos no se reflejan cambios debido a que no contienen ninguna entidad.

El proceso de revisión continua visitando nodos (reflejando los cambios ocurridos en aquéllos nodos que contienen entidades) hasta que se alcanza el nodo donde se inicio la revisión (el nodo activado) luego de completar un ciclo completo sin movimientos de mensajes.

Una vez que la revisión termina, el simulador busca otra entrada de la lista de evento futuros (FEL), para determinar el próximo evento en ser ejecutado. La FEL es, por supuesto, una colección de eventos: etiquetas de los nodos ordenadas según el tiempo fijado para su ejecución.

Éste es el algoritmo general de GLIDER. GALATEA, conserva esta estrategia general, excepto que, basado en la especificación proporcionada por la teoría de simulación multi-agentes [Dávila–Tucci:2000], proporciona la ejecución de un motor de inferencia asociado a cada agente. Todos estos motores se ejecutan concurrentemente con el simulador principal y su ejecución se intercala cuidadosamente, para intercambiar información y permitir la simulación de un sistema multi-agentes en forma efectiva. Los motores de inferencia proporcionan la simulación del ciclo percibir-razonar-actuar de cada agente mientras el simulador controla la evolución física del sistema.

Este enfoque en el que la simulación de un sistema se realiza a partir de los componentes distribuidos no es una idea nueva. De hecho, como mencionamos anteriormente, estamos aprovechándonos de una especificación existente para simulación distribuida, incorporando la simulación en una federación. El estándar HLA especifica las condiciones mínimas que un conjunto de componentes debe cumplir ser convertidos en una federación. En nuestro caso, el simulador y cada uno de los agentes es un federado. Ellos se agrupan en una federación multi-agentes que usa una estrategia simple, centralizada, de manejo de tiempo para controlar su propia evolución. En particular, en GALATEA los intercambios de información son siempre actualizaciones de la lista de eventos futuros, FEL, (o conjunto de influencias). Los agentes incorporan sus intenciones (acciones que quieren ejecutar) en la FEL y el simulador decide el desenlace de las mismas. De igual forma, el simulador determina y programa las percepciones apropiadas para cada agente en la FEL, y cada agente decide lo que

quiere ver.

Este algoritmo general para GALATEA, ha permitido especificar los detalles que deben regir el comportamiento del simulador. A continuación daremos una breve descripción del comportamiento del simulador.

1.1.3 Comportamiento del simulador

A continuación mencionamos los componentes asociados al proceso de ejecución en la plataforma GALATEA de una simulación que incluye agentes a lo largo del tiempo entre los objetos que participan, destacándose las actividades recursivas de activación y recorrido de la red de nodos y la activación de los agentes que participan en el sistema. Los componentes asociados al proceso de simulación son:

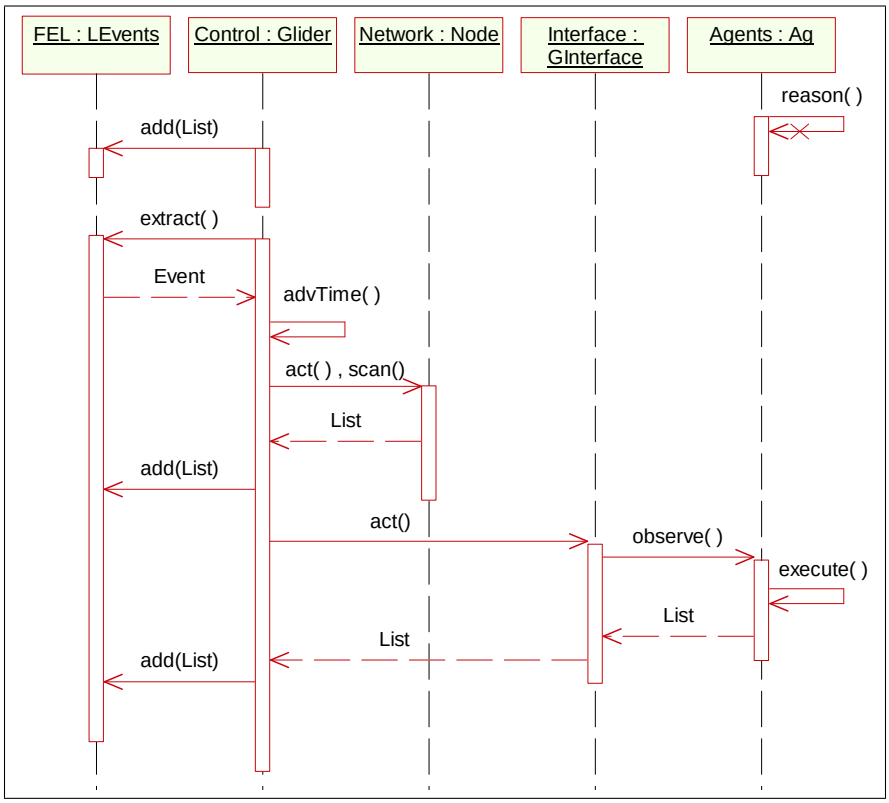

Figura 1.2: Diagrama de secuencia para GALATEA

FEL

Contiene, cronológicamente ordenados, los eventos que deben tener lugar en la simulación del sistema. El objeto FEL realiza las siguiente operaciones:

1. incluye en la lista de eventos los eventos que recibe,
2. si se le solicita, entrega el evento que debe ser procesado.

Control

Representa el controlador del proceso de simulación realizando las siguientes actividades:

1. solicita la programación de los eventos que permiten el inicio del proceso de simulación. Este procesamiento incluye la activación de la red de nodos y de los agentes,
2. solicita el próximo evento y controla que se procese el evento,
3. programa los nuevos eventos,

Network

Representa el conjunto de subsistemas (nodos) presentes en el sistema a simular y por lo tanto, tiene como función controlar la interacción y activación de los nodos. Este componentes debe cumplir las especificaciones descritas en la sección 2.1 de [Uzcátegui:2002] y debe llevar a cabo las siguientes actividades:

1. controla la activación del nodo actual,
2. controla la revisión de la red,
3. controla la activación de los agentes.

Interface

Sirve de intermediario entre los agentes y el simulador. Este componente debe cumplir con las pautas descritas en la sección 3.1 de [Uzcátegui:2002].

Agents

Representa al grupo de agentes presentes en el sistema. Este componente tiene la función de controlar las actividades propias del agente:

1. razonar. Implica asimilar percepciones, observaciones, recuerdos, metas, creencias y preferencias para tomar decisiones sobre que afectarán su conducta,
2. observar. Percibir el ambiente,
3. actuar. Intentar modificar el ambiente de acuerdo a las decisiones tomadas.

1.2 Agentes

Desde hace varios años hemos podido observar como ha crecido vertiginosamente la inteligencia artificial, con el surgimiento de nuevas técnicas computacionales que aprovecha otras disciplinas como la filosofía que involucra teorías del razonamiento y del aprendizaje; de las matemáticas aprovecha las teorías formales relacionadas con lógica, probabilidad, toma de decisiones y computación. De la psicología toma las herramientas que permiten la investigación de la mente humana, así como el lenguaje científico para expresar las teorías que se van obteniendo. La lingüística le ofrece teorías sobre la estructura y significado del lenguaje. Y por último de la ciencia de la computación, toma las herramientas que permiten que la inteligencia artificial sea una realidad, todas estas disciplinas han permitido la construcción de programas autónomos, adaptativos, capaces de aprender de las experiencias y de adaptarse a nuevas situaciones permitiendo realizar actos por si solos y dependiendo de la situación dada pueden emprender cualquier tipo de acción, de manera que en muchos de los casos no sabemos si la acción realizada fue ejecutada por un ser humano o por una maquina. En líneas generales estos programas los podemos denominar agentes, y aunque pueden emprender algunas acciones no pueden sustituir en su totalidad todas las tareas realizadas por un ser humano. Sabemos que en muchos de los casos se está trabajando para que estos al menos puedan emprender acciones de manera racional.

1.2.1 Definición de Agentes

Según Russell-Norvig [Russell–Norvig:1996], un agente es todo aquello que sea capaz de percibir su ambiente mediante sensores y que responde y actúa en tal ambiente por medio de efectores.

En el caso de nosotros los humanos nuestros sensores son los ojos, oídos y otros órganos, mientras que nuestros efectores podrían ser manos, piernas, boca y otras partes del cuerpo. En el caso de los agentes de software, donde nosotros nos centraremos sus percepciones y acciones vienen a ser la cadenas de bits codificados que reciben y envía el programa.

En la figura 1.3 observamos un modelo muy simple de un agente, y la característica más importante que estos poseen es la forma de capturar lo que esta ocurriendo en el ambiente por medio de los sensores y una manera de causar cambios a través de sus efectores. Existe una dinámica entre los sensores y los efectores y el trabajo actual de muchos investigadores se concentra en esta dinámica, como es, como se construye, como se le puede programar si es un programa, como se le puede cablear si es simplemente hierro. Entre los detalles mas interesantes que tenemos en estos agentes, es que los sensores son más efectivos que nuestros sentidos humanos, cámaras que pueden ver detalles que nosotros no podemos capturar, micrófonos que pueden

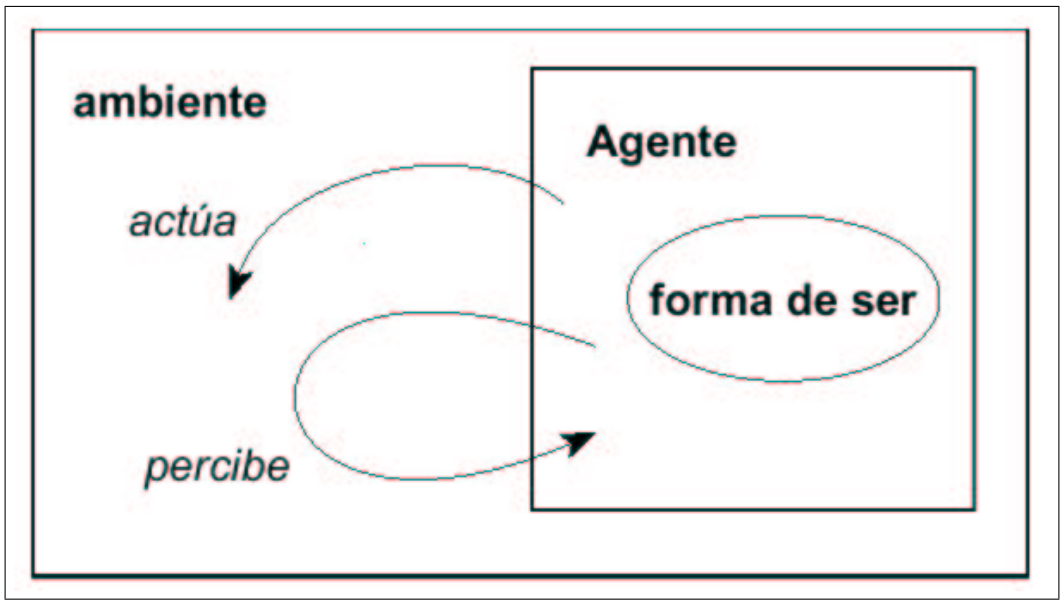

Figura 1.3: Estructura de un agente

escuchar cosas que nosotros no podemos escuchar y además tenemos efectores mucho más efectivos que los nuestros, mucho más poderosos, mucho más fuertes, resistentes a condiciones ambientales mucho más extremas, es decir; por el lado de construir el agente en cuanto a la construcción de hierro, pareciera estar bastante adelantado, el asunto es de integración y es que conectar los mecanismo que utiliza para percibir con los que utiliza para actuar se ha vuelto muy difícil, claro hay que tener una incisión de lo que hay en nuestro caso entre la percepción y la ejecución de acciones, es que hay un estado interno, hay una mente, dicen unos en el camino, hay que meterles un programa, ese software es la mente, leen(percibe) cosas que es la entrada , el programa muestra(actúa) una salida y unos podrían considerar que el asunto fue resuelto.

Este ejemplo de la figura 1.4 de Russell-Norvig[Russell–Norvig:1996] es de hecho el primer ejemplo que muestra estos autores en el libro Inteligencia Artificial, este modelo practico es el más elemental, yo tengo los sensores y los efectores, ellos me reportan la información que me aparecen en los cuadros, como es el mundo ahora me dicen los sensores, que debo hacer es lo que yo le comunico a los efectores, los conectan por medio de una tabla un sistema que en inteligencia artificial es llamado un conjunto de reglas de condición-acción. Una lista de condiciones que están puestas en una tabla y una lista de acciones que normalmente es una sola acción que están en columnas contiguas por nosotros y entonces yo reviso cuales son las condiciones que se cumplen en el ambiente, disparo una acción y funciona.

Este es un agente que le llaman reflectivo o que funciona por reflejos simples y en mucho de los casos no tiene la capacidad de medir las consecuencias que pudieran

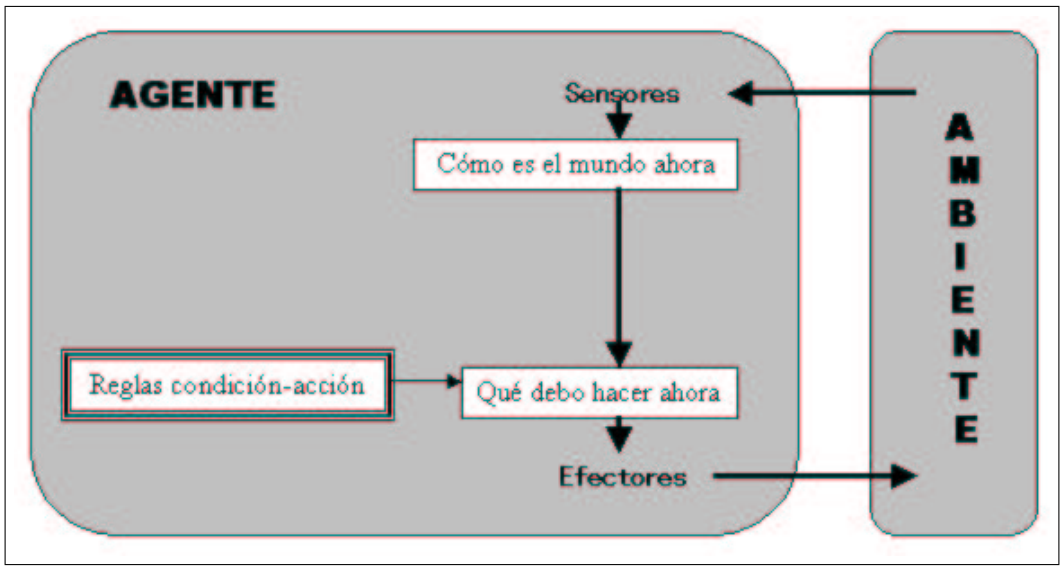

Figura 1.4: Modelo de agente según S. Russel. y P. Norvig

generar sus acciones. Y está resuelto el problema, el asunto es que esa dinámica interna basada en el conjunto de reglas de condición y acción se queda corta muy rápida mente, el asunto es cuando queremos que el agente haga algo que tenga un poco más de sentido humano, nos vemos en problemas para tratarlo de codificarlo con reglas de condición y acción, incluso hay estudios sistemáticos de lo que significa codificar cualquier aplicación más o menos regular en términos de regla de condición y acción , son demasiadas reglas, sin contar con el esfuerzo de escoger cual es la que sirve, entonces este modelo sirve para ver cuales son los componentes esenciales, pero no nos podemos engañar de que el cuerpo del agente y el ambiente son cajitas, estas cajitas pueden ser software, puede ser intangible, o simplemente ser una sensación digital y esta separación psicológica-psiquiátrica con la cual separamos la mente y cuerpo del agente hay que manejarla con mucho cuidado; entonces podemos hablar de que el ambiente puede ser solamente software por ejemplo y lo que estamos programando es la mente, el controlador del agente; y un detalle bien importante referir el agente a algo que nosotros conozcamos, referir a funciones matemáticas que se supone que todos los ingenieros saben algo de eso, y es por eso que el agente de la figura 1.4 se puede representar de esta manera, con modificaciones realizadas por el Profesor Jacinto Dávila presentadas en el Tutorial sobre Agentes en la XXVII CLEI (Conferencia Latinoamericana de Informática) y lo represento de esta forma:

function Simplex-Reflex-Agent(*Percept* percepto)
static reglas, *%conjunto de reglas de condición-acción.*
estado ← intepretaEntradas(percepto);
regla ← seleccionaRegla(estado, reglas);

accion ← calculaAccion(regla);
return accion;

De esta manera queda más claro por medio de funciones matemáticas el funcionamiento del agente o por lo menos el agente reactivo, este recibe un solo precepto una sola unidad de información de entrada y devuelve una acción y lo que está en itálica y en mayúscula es el tipo de dato que esta pasando. El precepto y las acciones se convierten en información. ¿Y que pasa en el medio?, bueno; hay otra función que interpreta el precepto y conduce a un estado interno nuevo, cambia el estado interno y se entera el agente de lo que esta pasando haya afuera y con esta información, con este nuevo estado tiene que leerlo en orden secuencial y con el conjunto de reglas que tiene permanente, programadas en alguna parte con información estática, escoge una de esas reglas para ver cual es la que se esta activando, la selecciona y con esta regla seleccionada decide que hacer. Trae la acción que justamente lo que hace la función calculaAccion, y eso es todo. Para poder de que estas funciones se convierta en un agente inteligente hay que por supuesto complicar esta estructura, pero eso es esencialmente la estructura misma.

1.2.2 Agentes Inteligentes

Son aquellos tipos de agentes que poseen autonomía, iniciativa e inteligencia propia por decirlo de alguna manera, según (McCarthy y Shohan) en IA se está estudiando para obtener agentes que exhiban características de la inteligencia humana y estos agentes se construyen según Rusell cuando:

- describimos su comportamiento y el ambiente mediante "conocimiento".
- dispone de iniciativa de explorar el ambiente.
- incrementa su conocimiento basándose en la experiencia.
- evalúa la consecución de las metas que se le plantean.

Agente Racional

Un agente racional es aquel que emprende acciones correctas a través de sus efectores en el ambiente donde se encuentra, debe tener un buen desempeño y a demás sus acciones deben ser autónomas.

La racionalidad es uno de los principales objetivos en Inteligencia Artificial actualmente. Según Norvig[Russell–Norvig:1996] si un agente hace lo correcto ofrece dos ventajas. La primera es más general que el enfoque de las "leyes del pensamiento"presentado por Aristóteles [Russell–Norvig:1996], dado que efectuar inferencias correctas es sólo un mecanismo útil para garantizar la racionalidad, pero no es un

mecanismo necesario. La segunda es más afín a la manera como se ha producido el avance científico que los enfoques basados en la conductas o el pensamiento humano, toda vez que se defina claramente lo que será la norma de la racionalidad, norma que es de aplicación general. Por lo contrario que la conducta humana se adapta bien sólo en un entorno y, en parte, es producto de un proceso evolutivo complejo y en gran parte innato, cuya perfección todavía se ve distante.

En resumen, el carácter de racionalidad de un agente en un momento dado dependerá de cuatro factores.

- De la medida con la que se evalúa el grado de éxito logrado.
- De todo lo que hasta ese momento haya percibido el agente.
- Del conocimiento que posea el agente del medio.
- De las acciones que el agente pueda emprender.

Cuando hablamos de la racionalidad de un agente hay un elemento más al que se debe prestar atención: la parte del conocimiento integrado. Si las acciones que emprende el agente se basan exclusivamente en un conocimiento integrado, con lo que hace caso omiso de sus percepciones, se dice que el agente no tiene autonomía.

La conducta del agente se basa tanto de su propia experiencia como en el conocimiento integrado que sirve para construir al agente para el ambiente específico en el cual va operar. Un sistema será autónomo en la medida en que su conducta está definida por su propia experiencia.

Otro punto importante que hay que tomar en cuenta es el termino medición de desempeño y se aplica al cómo: es el criterio que sirve para definir que tan exitoso ha sido un agente. Desde luego no hay una medición fija que se pueda aplicar por igual a todos los agentes. Para realizar esta medida, se debe contar con una autoridad, un observador externo que permita definir las normas de lo que se pudiera considerar un buen desempeño en el ambiente y emplearlo en la medición de desempeño de los agentes.

Agentes sociales o cooperativos

Son aquellos agentes que realizan sus actividades en conjunto y funcionan de la siguiente manera:

1. Se recibe un problema en cierto nivel de abstracción.
2. El agente resuelve localmente aquello que es posible.
3. Recorre a otros agentes del mismo nivel para el resto de las tareas.

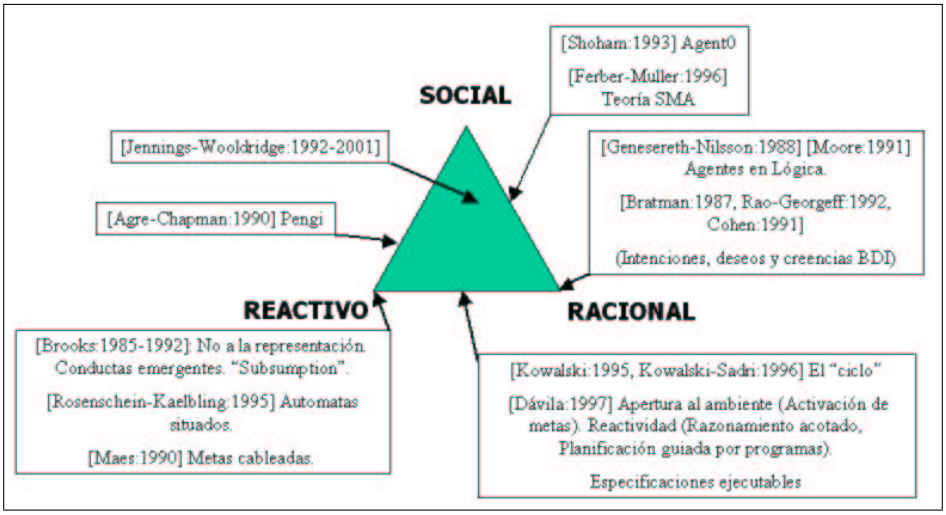

Figura 1.5: El triángulo reactividad-racionalidad-sociabilidad

4. Recorre a otros niveles de abstracción para el resto de las tareas.

En la figura 1.5 mostramos en triple dimensión la sociabilidad, racionalidad y reactividad de un agente a través de un triángulo presentado por el profesor Jacinto Dávila en el CLEI 2001. Hay una dimensión, el componente reactivo, que asegura que nuestro agente haga algo, que lo haga algún día, debe actuar oportunamente. Esta la dimensión racional que es la que asegura que el agente haga algo correctamente, inteligentemente y esta la dimensión social que debe asegurar que el agente pueda cooperar con otros agentes sin hacer algo, no son por su puesto una justificación fácilmente que permita distribuirse en una fase social, pero funciona bien para el trabajo que se esta haciendo, por ejemplo nosotros vamos a hablar de agentes inteligentes y hay tenemos agentes que son capaces de hacer manipulaciones simbólicas por medio de un manipulador simbólico que asocia a su entrada con la salida de manera inteligente; eso es lo que lo va hacer racional y esa manipulación simbólica es una manipulación de forma lógica, fue una propuesta realizada por el profesor Jacinto Dávila[Dávila:1997] y la puso en el medio porque la otra posición que obtuvo siguiendo una propuesta del profesor Kowalski, es que con el cuidado con el que puede estar diseñado el agente, además de ser racional sea reactivo, entonces este primer agente esta en la mitad más o menos del eje reactivo-racional, pero hay propuestas que se preocupan que el agente pueda interactuar con el entorno social inteligentemente y entonces desde 1990 se esta hablando de la programación orientada a los agentes, una propuesta del profesor Shohan que aparece formalizada en 1993 con la propuesta del Agente0 que esta ubicado en el eje social-racional, un agente que esta apuntando como actúa un agente correctamente en sociedad, tal como lo vemos en los sistemas orientado a objetos que son una colectividad pequeña de quantum, pero además; todo

este escándalo de agentes esta asociado a una serie de puertos para mostrar como se estaba haciendo antes, el concentrarnos en el aspecto racional del asunto era, ir en la dirección equivocada. La propuesta del profesor Brooks de la IMT dice que nos olvidemos completamente de la representación del conocimiento, fue algo rebelde de su parte según el Profesor Dávila, pero que argumento y además mostró sistemas sin ninguna clase de enfrentamiento simbólico que se comportaban muy bien, sobre todo muy bien; en términos de acciones de sistemas que se requieren intervalos de respuestas muy rápidos, por ejemplo andar por ahí sin tropezar con ningún obstáculo, capturar una pelota en el aire y todas esas clase de conductas que tienen que ver con actuar rápidamente, el asunto con él y esa fue la razón de dejar ese proyecto tratando de reconciliar reactividad con racionalidad es que el sistema no escale, hacer un sistema por ejemplo razonar a más largo plazo, sobre más largos periodos de tiempo, allí parece fundamental el uso de algún tipo de representación y por eso la intención de volver del Profesor Dávila de tratar de combinar la reactividad con la racionalidad. Bueno para el profesor Dávila y su equipo lo que quiere es estar en la mitad del triángulo, y bueno hay dos respuestas para esto; la una es la respuesta practica de que yo quiero resolver mi problema, tengo aquí un sistema que los agentes me lo manejen, entonces aquí tengo ya que pensar cual de los ejes es el que más interesa y sacar la cuenta de compromiso probablemente yo quiero una cosa que funcione rápidamente la pongo muy cerca del componente reactivo, pero con un nivel de racionalidad y un nivel de interacción social o yo quiero algo que al mundo interaccione socialmente la pongo cerca del componente social, este es una de las respuestas. La otra respuesta es la que hacemos en la academia de eso; que es de entender el problema completo, cuales son los compromisos de moverse por toda el área del triángulo, ahora habría que caracterizar el espacio completo y poder ofrecerles al que utiliza las aplicaciones tácticas el esquema de evaluaciones, a tal aplicación poderle decir tiene que hacer tal cosa, con tal herramienta de desarrollo, lo que se quiere es caracterizar el sistema completo. Bueno es pensar que se comparte inteligentemente los tres componentes y mucha gente piensa que de ahí puede salir o puede haber comportamiento humano, es decir; algo cojo de inteligencia humana, pero como ingenieros nos obligaría a filosofar.

En la figura 1.6 presentado por el profesor Jacinto Dávila en el CLEI 2001, observamos diversos marcos teóricos sobre agentes. Y podemos decir que el conocimientos que estos poseen no proviene solamente de filosofía, sino que proviene de una gama de disciplinas, que mencionamos al principio de esta sección, o como por ejemplo la teoría de control cibernética que explica con mucho entusiasmo el funcionamiento de los robots. La psicología y teorías del funcionamiento de la motivación, del porque los agentes hacen lo que hacen y como hacemos nosotros para producir esas condiciones dentro de un dispositivo. Teoría de la economía y teorías de las organizaciones y sociología, posiblemente el primer modelo de agentes se lo debemos a un economista a Simon, que curiosamente construyó algo parecido al agente presentado en la figura

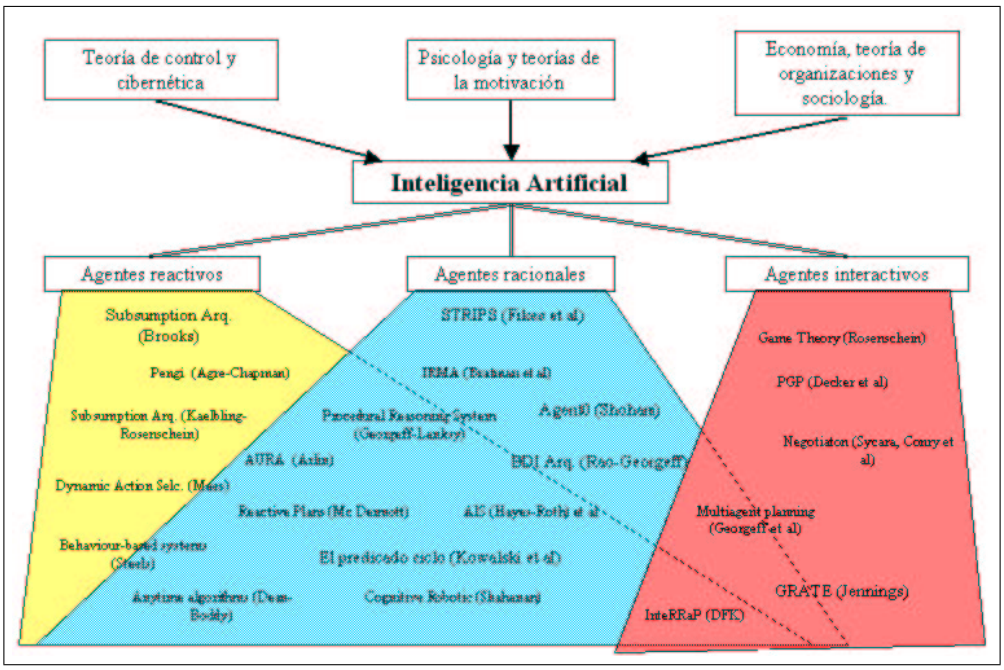

Figura 1.6: Diversos marcos teóricos para los agentes. Contextos de estudio (DFKI, Dr. K Fischer)

1.4.

Estas teorías son la inspiración de los proyectos que se adelantan en Inteligencia Artificial, y en que apuntan; a los tipos de agentes que mencionamos antes, solo que aquí cambiamos lo social por lo de interactivo, quizás explica más el por que nos paramos en uno solo para hablar de social, la intención es hacerlo interactivo, aquí esta una lista de nombres famosos ubicados más o menos en las áreas vinculados de lo que hacen, observamos en la figura 1.6, proyectos que realizan esfuerzos para mezclar dos cosas, para que sus agentes tengan algo de reactivo y racional, o que posean algo de racional e interactivos, si nos fijamos el esfuerzo del proyecto Multiagent planning (Georgeff el al), el cual no se procesa en el área de interactivo puro, no esta en el área de ser social, para interactuar con el ambiente. Por ejemplo el esfuerzo del proyecto Alemán (INTERRAP (DFE)) esta pensado para combinar las tres cosas y para más información se puede conseguir información en internet al respecto, hay cosas como Teorías de Juego pensado en interactividad. El proyecto Dynamic Actión Selc, de la IMT, pensados en agentes que hagan cosas útiles, que sean capaces de resolver cosas oportunamente, esta es una revisión muy general.

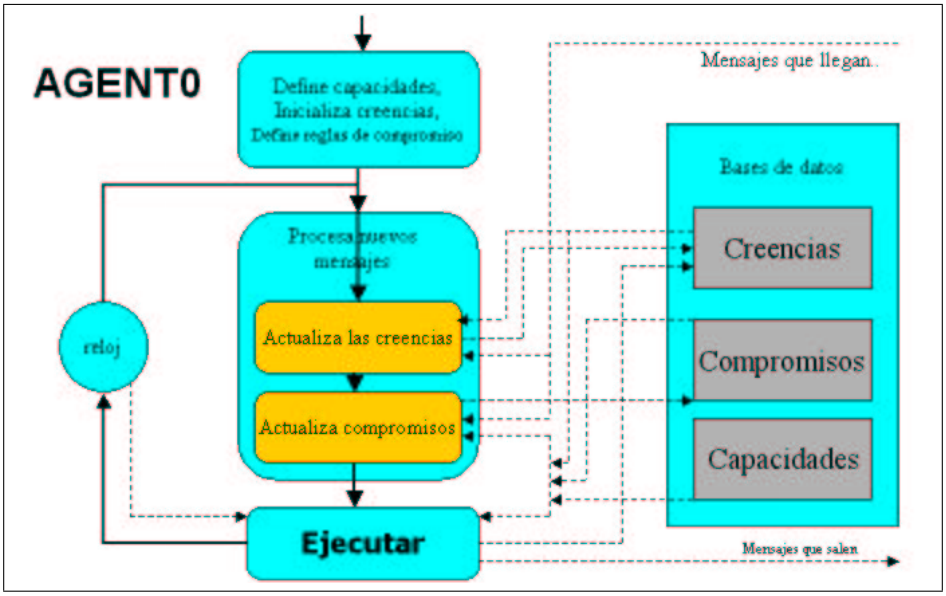

Figura 1.7: Agentes inteligentes según Shoham

Un modelo de agentes inteligentes

En la figura 1.7, observamos el modelo de Agentes según Shohan presentado por el profesor Jacinto Dávila en el CLEI 2001, este agente está representado por medio de cajitas, alguna de ellas; las cajitas con la punta curva representan procesos, procesos andándoos y la cajitas con las puntas en forma de cajas representan datos, estructuras de datos esencialmente, estructuras de conocimientos en este caso. ¿Y que es lo que tenemos?, bueno; lo que tenemos es un objeto, el agente es un objeto, que reciben mensajes y que emiten mensajes igualito que la programación por objeto, quizá un poquito más allá, porque aquí la semántica de los mensajes, se aproxima a lo que nosotros entendemos por mensajes y no lo que nos obliga la programación por objetos a entender por mensajes, que es una orden. No, aquí me mandan un mensaje que es el proceso interno que se va encargar de digerir, me pidieron tal cosa, un momentito (dice el agente) puedo hacer la cosa que me pidieron, entonces yo chequeo mis capacidades para decidir si puedo hacerlas y si puedo hacerlas, no solo chequeo mis capacidades, sino chequeo como lo voy a hacer de acuerdo con mis creencias, haber como esta el mundo, haber si es; como las puedo hacer, estas cosas interactúan y si puedo hacerlas es posible que prometa hacerlas, ese es el compromiso. Entonces la estructura del profesor Shohan se parece mucho al trabajo del profesor Dávila [Dávila:1997]. Hay una base de conocimiento, que en este caso es la creación de creencias y capacidades y hay una serie de metas, esas es la cosa que yo quiero hacer, que es el compromiso; y

el compromiso son en el esquema de Shohan muy parecidos a las reglas de condición-acción en [Dávila:1997], el profesor Shohan lo que hizo es complicar la representación condición-acción utilizando un tipo de lógica diferente, pero en líneas generales, esto es un agente. Donde tiene que haber un proceso inicial para su construcción, se le pone al azar y el va ha estar iterando al inicio de las creencias de lo que llegan de fondo que son siempre mensajes de otros agentes por eso es un agente social y no reactivo, actualiza los compromisos que son los compromisos que yo estoy incorporando, en este caso ejecuta las acciones a los compromiso que tiene por allí pendiente y le da la vuelta marcando un nuevo ciclo en el reloj.

1.2.3 Otros tipos de Agentes

Según sea el uso de los agentes o a la forma como los agentes realizan sus acciones los agentes lo podemos clasificar como: agentes reactivos (mencionado anteriormente), agentes basados en metas y agentes basado en utilidad[Russell–Norvig:1996].

Agentes basados en metas

Los agentes basados en metas, son aquellos que para decidir que hay que hacer no se bastan con la información acerca del estado que prevalece en el ambiente, si no la decisión adecuada depende de la meta que este posee. Es decir; además de una descripción del estado prevaleciente, el agente también requiere de cierto tipo de información sobre su meta, información que detalle las situaciones deseables. El programa de estos tipos de agentes, es poder combinar su meta con la información relativa al resultado que producirían las posibles acciones que se emprendan y de esta manera elegir aquellas acciones que permitan alcanzar la meta. En ocasiones esto es sencillo, cuando alcanzar la meta depende responder con una sola acción; otras veces es más complicado, cuando el agente tenga que considerar largas secuencias de acciones para alcanzar la meta. Hay que tomar en cuenta que la decisión de este tipo de Agentes difiere radicalmente de las reglas condición-acción, ya que implica el tomar en cuenta el futuro. En el caso del diseño del Agente reactivo esta información no se utiliza explícitamente puesto que el diseñador calcula la acción correcta correspondiente a diversos casos. Si bien el agente basado en metas es menos eficiente que el agente reactivo a la hora de responder, también es cierto que estos son más flexibles que los reactivos.

En la figura 1.8, se muestra la estructura del agente basado en metas, los sensores le muestra el ambiente, al estado interno del agente; como es el mundo ahora, como evoluciona el mundo, el agente revisa en su base de conocimiento reglas de condición-acción relacionadas con el mundo actual, y antes de emprenderla, se pregunta: ¿Qué sucedería si emprendo esta acción?, revisa sus metas y de acuerdo a estas ejecuta la más conveniente a través de us efectores.

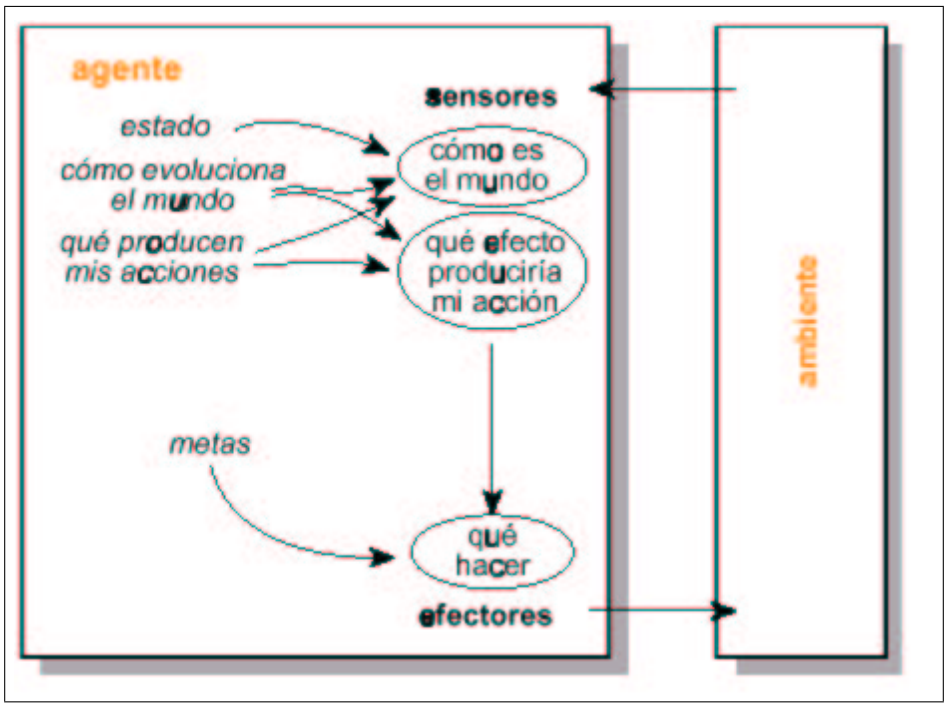

Figura 1.8: Agente basado en metas

Agentes basados en utilidad

Los agentes basados en utilidad permiten tomar una decisión racional en dos tipos de casos en el que los agentes basados en metas se encuentran en problemas. El primero, cuando el logro de algunas metas implica un conflicto, y sólo algunas de ellas se pueden obtener, la función de utilidad definirá cuál es el compromiso adecuado por el cual hay que optar. Segundo, cuando varias son las metas que el agente podría desear obtener, pero no existe la certeza de poder lograr ninguna de ellas, la utilidad es una vía para ponderar la posibilidad de tener éxito considerando la importancia de las diferentes metas.

Un agente que posee una función de utilidad explícita puede tomar decisiones racionales, aunque tenga que comparar la utilidad con diversas acciones.

Mientras que los agentes basados en metas, su inflexibilidad, le permite optar de inmediato por una acción cuando este puede alcanzarla. Además, en algunos casos, se puede traducir la función de utilidad en un conjunto de metas, de manera que las decisiones adoptadas por el agente basado en metas tomando como base tales conjuntos de metas resulte idénticas a las que haría el agente basado en utilidad.

En la figura 1.9, se muestra la estructura general de un agente basado en utilidad, y a diferencia de la figura 1.8, es que luego de revisar sus metas y escoger la acción a realizar, verifica que tan a gusto encontrará el estado al ejecutar la acción seleccionada

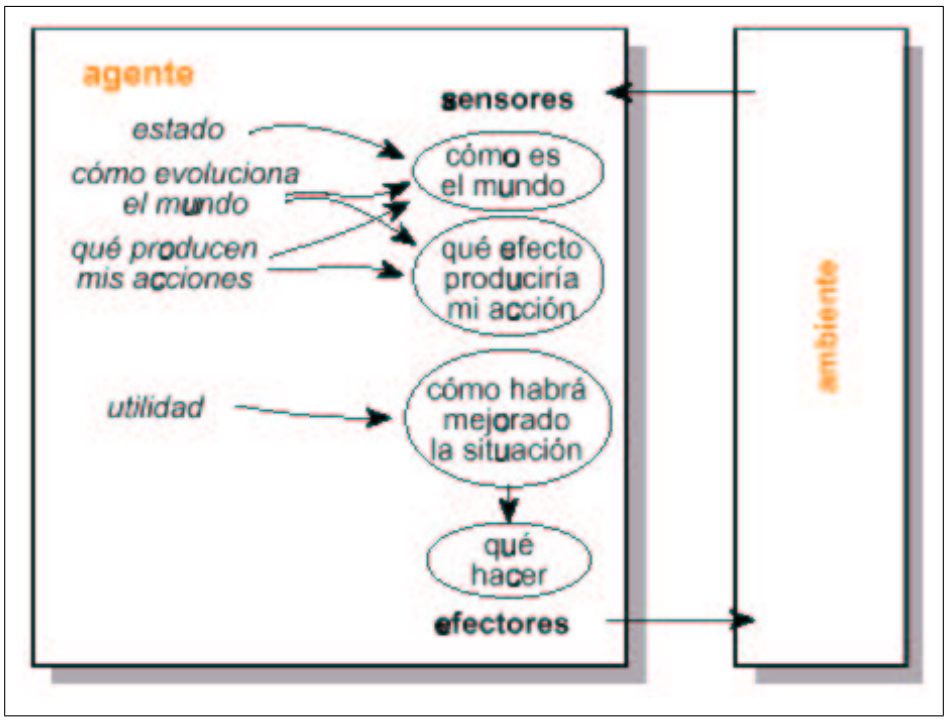

Figura 1.9: Agente basado en utilidad

y así poder tener un máximo de felicidad en el ambiente.

1.2.4 Arquitectura y funcionamiento de los agentes

Hasta ahora nos hemos centrado en la conducta que debe tomar una agente ante cierta situación y si esta conducta tomada es la mejor o no, para abordar un problema y encontrar la mejor solución, también hemos hablado como se puede clasificar los agente según las acciones que estos emprenden pero no hemos hablado de la parte interna del agente.

El agente internamente debe poseer un motor de inferencia que le permite incorporar un mapeo de todas las secuencias de percepciones tomadas en un cierto tiempo en su base de conocimiento para realizar la acción o las acciones pertinentes según sea el caso. Este motor de inferencia se ejecutará en algún tipo de dispositivo de cómputo, al que se le denominará arquitectura o interfaz.

La arquitectura pone al alcance del motor de inferencia las percepciones obtenidas de los sensores, las ejecuta y alimenta al efector con las acciones elegidas dentro de su base de conocimiento conforme éstas se van generando. La relación entre la arquitectura y los programas se podría resumir de la siguiente manera

Agente = arquitectura + programa(motor de inferencia + base de conocimiento)

Una vez que nos hemos dado cuenta del comportamiento de un agente depende exclusivamente de las secuencias de sus percepciones en un momento dado, sabemos que es posible caracterizar cualquier agente en particular elaborando una tabla con las acciones que este emprende (base de conocimiento) como respuesta a cualquier secuencia de percepciones posibles.

En muchos tipos de agentes su base de conocimientos se actualizan con la llegada de nuevas percepciones. Esta base de conocimientos son estructuras de datos que operarán mediante los procedimientos de toma de decisiones (que pueden ser programados por medio lenguajes de programación lógico) a través de reglas, para generar la elección de una acción por medio de su motor de inferencia, esta acción se transferirá a la arquitectura que sirve de efector para efectuar la acción.

El mapeo del agente permite recibir las secuencias de percepciones que vendrán codificadas en nuestro caso en Java (que será el canal de interpretador Prolog), para buscar cual es la acción o secuencias de acciones a tomar ante las secuencias de percepciones recibidas en cierto instante.

Para realizar el agente se debe comenzar primero por realizar la tabla de consulta (base de conocimiento, que consiste en guardar una serie de reglas). Se guarda en memoria la serie de percepciones que son percibidas mediante los sensores y utilizar estas para realizar la consulta en la base de conocimiento por medio de su motor de inferencia y tomar la acción para todas las posibles secuencias de percepciones.

Se debe tomar en cuenta que si las acciones que debe emprender el agente se basan exclusivamente en un conocimiento integrado, haciendo caso omiso de sus percepciones, se dice que el agente no tiene autonomía.

La conducta de un agente se basa tanto en su propia experiencia como en el conocimiento integrado que sirva para construir al agente específico en el cual va operar. Un sistema será autónomo en la medida en que su conducta está definida por su propia experiencia, además de dotarlos de inteligencia artificial con ciertos conocimientos iniciales de capacidad para aprender.

Ahora realizando una abstracción con respecto a la forma de abordar la construcción de un agente de acuerdo a las acciones que estos emprenden pueden ser de la siguiente manera:

Arquitectura reativa

- Estas arquitecturas manejan jerarquías de tareas en función de niveles de abstracción.
- Un razonamiento explícito sobre los efectos producidos por acciones de bajo nivel es demasiado costoso para producir una conducta en tiempo real.
- El comportamiento inteligente se generará sin tener que usar modelos simbólicos,

y emergerá en ciertos sistemas complejos[Shoham:1990].

Por ejemplo

- Agentes reactivos.
- Agentes con representación del mundo.

Arquitectura deliberativa

Expresa el comportamiento y el ambiente en términos de conocimiento, que es representado simbólicamente y donde las decisiones se toman empleando mecanismos deductivos.

En estos términos, los componentes del agente se deben representar en términos lógicos. Un ejemplo exitoso: BDI (Belief, Desire, Intention)[Rao, Georgeff], donde:

- Creencias(Belief): modelo del mundo y del resto de agentes.
- Deseos(Desire):metas.
- Intenciones(Intention): plan de acción.

Como ejemplo de este tipo de arquitectura, podemos nombrar a:

- Agentes basados en Metas.
- Agente basados en utilidad.

Arquitecturas híbridas

Combinan agentes de tipo reactivo y deliberativo. En el caso de los agentes reactivos estos reacciona a los eventos del entorno sin invertir razonamiento. Mientras que los agentes con arquitectura deliberativa, planifica distribuyendo las metas más simples y realizan tareas de nivel de abstracción superior. Este tipo de tareas se agrupan en dos niveles; el primero es de manipulación de la información a nivel abstracto y contiene una representación simbólica del ambiente, el segundo es de comportamiento social y planificación de alto nivel.

Esta tipo de arquitectura se puede organizar horizontalmente cuando la capas tienen acceso a sensores y actuadores, o verticalmente cuando una capa actúa de interfaz con sensores y actuadores.

1.2.5 Agentes y objetos

Los principales conceptos que definen la orientación a objetos son: relación clase/instancia, herencia y mecanismos de mensajes; polimorfismo.

Por otro lado los principales conceptos que definen a un agente son: autonomía permite que se pueda ejecutar procesos en paralelo, iniciativa por lo cual el agente la utiliza para perseguir sus metas y mecanismos de comunicación de alto nivel.

La POA (Programación Orientada a gentes, puede considerarse como una especialización del paradigma de la POO (Programación Orientada a Objetos)[Shoham:1990]; entones un objeto puede considerarse cierto tipo de agente simplificado y en la figura 1.10 mostramos una extracción de los objetos y los agentes.

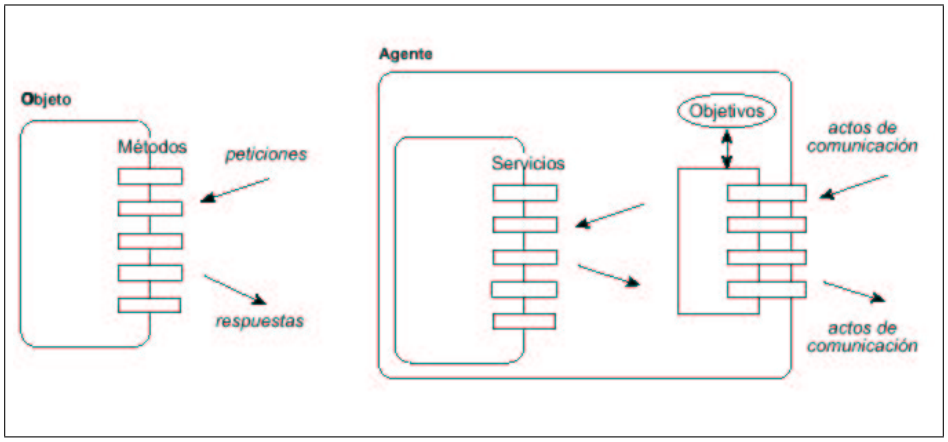

Figura 1.10: Agentes y objetos

Modelado orientado a agentes

Los mecanismos de abstracción del modelo conceptual usual en POO se aplican en POA:

- clasificación / instanciación.
- agregación / descomposición,
- generalización / especialización,
- agrupación / individualización.

La POO incorpora su acervo a las futuras metodológicas de desarrollo de agentes.

1.2.6 Sistemas Multiagentes

Los Sistemas multi-agentes (MAS: Multi Agent System), son un conjunto de agentes autónomos, generalmente heterogéneos y potencialmente independiente que trabajan en común resolviendo un problema.

Las características de estos Agentes vinculados con la noción de agente inteligente es que puedan realizar tareas como:

- capaz de tomar iniciativa,
- capaz de compartir conocimiento,
- capaz de cooperar y negociar,
- capaz de comprometerse con metas comunes.

Y además debemos acotar que el estudio de MAS se encuentra dentro de la Inteligencia Artificial Distribuida (DAI).

Inteligencia Artificial Distribuida

Es una rama de la IA que estudia la solución de problemas mediante procesamiento descentralizado.

- Resolución distribuida de problemas (DPS):

  Se descompone el problema en procesos cooperantes que comparte conocimiento, para un problema concreto.

- MAS
- Inteligencia Artificial Paralela (PAI):

  Desarrollo de algoritmos y aplicaciones paralelas, con énfasis en prestaciones.

Agentes Inteligentes vs. Agentes no Inteligentes

- Agentes no inteligentes
  - Ausencia de estructura social entre agentes: interacción predefinida.
  - Universo estable, predecible y observable.
  - Consecuencia: sistemas pocos flexibles frente a entornos cambiantes.
- Agentes Inteligentes
  - Actúan en entornos cambiantes, inciertos e impredecibles.

  - Actúan con conocimiento incompleto.
  - Pueden cooperar para resolver problemas más complejos.

1.3 Representación del Conocimiento

En forma natural, el ser humano representa el conocimiento simbólicamente: imágenes, lenguaje hablado y lenguaje escrito. Adicionalmente, ha desarrollado otros sistemas de representación del conocimiento: literal, numérico, estadístico, estocástico, lógico.

La ingeniería cognoscitiva ha adaptado diversos sistemas de representación del conocimiento que, implantados en un computador, se aproximan mucho a los modelos elaborados por la psicología cognoscitiva para el cerebro humano. Entre los principales se tienen:

- **Lógica Simbólica Formal:**
  - Lógica proposicional
  - Lógica de predicados.
  - Reglas de producción.
- **Formas Estructuradas:**
  - Redes asociativas.
  - Estructuras marco.
  - Representación orientada a objetos.

1.3.1 Lógica Proposicional

La lógica proposicional es la más antigua y simple de las formas de lógica. Utilizando una representación primitiva del lenguaje, permite representar y manipular aserciones sobre el mundo que nos rodea. La lógica proposicional permite el razonamiento, a través de un mecanismo que primero evalúa sentencias simples y luego sentencias complejas, formadas mediante el uso de conectivos proposicionales, por ejemplo Y (AND), O (OR). Este mecanismo determina la veracidad de una sentencia compleja, analizando los valores de veracidad asignados a las sentencias simples que la conforman.

Una proposición es una sentencia simple que tiene un valor asociado ya sea de verdadero (V), o falso (F). Por ejemplo:

$$Hoy\ es\ Viernes$$
$$Ayer\ llovió$$
$$Hace\ frío$$

La lógica proposicional, permite la asignación de un valor verdadero o falso para la sentencia completa, no tiene facilidad para analizar las palabras individuales que componen la sentencia. Por este motivo, la representación de las sentencias del ejemplo, como proposiciones, sería:

$$hoy\_es_V iernes$$
$$ayer\_llovió$$
$$hace\_frío$$

La proposiciones pueden combinarse para expresar conceptos más complejos. Por ejemplo:

$$hoy\_es\_Viernes\ y\ hace\_frío.$$

A la proposición anterior dada como ejemplo, se la denomina **fórmula bien formada** (well-formed formula, wff). Una fórmula bien formada puede ser una proposición simple o compuesta que tiene sentido completo y cuyo valor de veracidad, puede ser determinado. La lógica proposicional proporciona un mecanismo para asignar valores de veracidad a la proposición compuesta, basado en los valores de veracidad de las proposiciones simples y en la naturaleza de los conectores lógicos involucrados.

1.3.2 Lógica Predicados

La principal debilidad de la lógica proposicional es su limitada habilidad para expresar conocimiento. Existen varias sentencias complejas que pierden mucho de su significado cuando se las representa en lógica proposicional. Por esto se desarrolló una forma lógica más general, capaz de representar todos los detalles expresados en las sentencias, esta es la **lógica de predicados**.

La lógica de predicados está basada en la idea de las sentencias realmente expresan relaciones entre objetos, así como también cualidades y atributos de tales objetos. Los objetos pueden ser personas, objetos físicos, o conceptos. Tales cualidades, relaciones o atributos, se denominan **predicados**. Los objetos se conocen como **argumentos** o **términos** del predicado.

Al igual que las proposiciones, los predicados tienen un valor de veracidad, pero a diferencia de las preposiciones, su valor de veracidad, depende de sus términos. Es decir, un predicado puede ser verdadero para un conjunto de términos, pero falso para otro.

Por ejemplo, el siguiente predicado es verdadero:

$$color(yerba, verde)$$

el mismo predicado, pero con diferentes argumentos, puede no ser verdadero:

$$color(yerba, verde) \; o \; color(cielo, \; verde)$$

Los predicados también pueden ser utilizados para asignar una cualidad abstracta a sus términos, o para representar acciones o relaciones de acción entre dos objetos.

Al construir los predicados se asume que su veracidad está basada en su relación con el mundo real. Naturalmente, siendo prácticos, trataremos que los predicados que definimos estén de acuerdo con el mundo que conocemos, pero no es absolutamente necesario que así lo hagamos. En lógica de predicados el establecer como verdadero un predicado es suficiente para que así sea considerado. En este ejemplo se indica que Ecuador está en Europa:

$$parte\_de(ecuador, europa)$$

Obviamente, esto no es verdadero en el mundo real, pero la lógica de predicados no tiene porque saber geografía y si el predicado es dado como verdadero, entonces es considerado como lógicamente verdadero. Tales predicados, establecidos y asumidos como lógicamente verdaderos se denominan **axiomas**, y no requieren de justificación para establecer su verdad.

La lógica de predicados, se ocupa únicamente de métodos de argumentación sólidos. Tales argumentaciones se denominan **Reglas de Inferencia**. Si se da un conjunto de axiomas que son aceptados como verdaderos, las reglas de inferencia garantizan que sólo serán derivadas consecuencias verdaderas.

Tanto los conectivos lógicos, como los operadores utilizados por la lógica proposicional, son igualmente válidos en lógica de predicados. De hecho, la lógica proposicional es un subconjunto de la lógica de predicados.

Cada uno de los argumentos en los ejemplos de predicados dados anteriormente, representan a un objeto específico. Tales argumentos se denominan *constantes*. Sin embargo, en la lógica de predicados se pueden tener argumentos que en determinado momento pueden ser desconocidos. Estos son los argumentos tipo *variable*. En el ejemplo:

$$color(yerba, X)$$

La variable X, puede tomar el valor de *verde*, haciendo que el predicado sea verdadero; o puede tomar el valor de *azul* dando lugar a que el predicado sea falso.

Las variables, también pueden ser cuantificadas. Los cuantificadores que típicamente se utilizan en lógica de predicados son:

- **El cuantificador universal**; $\forall$ indica que la fórmula bien formada, dentro de su alcance, es verdadera para todos los valores posibles de la variable que es cuantificada. Por ejemplo:

  $\forall X...$
  Establece "que para todo X, es verdad que ..."

- **El cuantificador existencial**; $\exists$, indica que la fórmula bien formada, dentro de su alcance, es verdadera para algún valor o valores dentro del dominio. Por ejemplo:

  $\exists X...$
  Establece que "existe un X, tal que ... "

Desde el punto vista de representación, los cuantificadores son difíciles de usar. Por lo que es deseable reemplazarlos con alguna representación equivalente, más fácil de manipular. El caso del cuantificador universal es más simple ya que se asume a todas las variables como universalmente cuantificadas.

El cuantificador existencial es más difícil de reemplazar. El cuantificador existencial garantiza la existencia de uno o más valores particulares *(instancias)* de la variable cuantificada, que hace a la cláusula verdadera. Si se asume que existe una función capaz de determinar los valores de la variable que hace la cláusula verdadera, entonces simplemente se remueve el cuantificador existencial y se reemplaza las variables por la función que retorna dichos valores. Para la resolución de problemas reales, esta función, llamada *función de Skolem*, debe ser conocida y definida.

Cuando se tienen sentencias compuestas por predicados y conectivos lógicos, se debe evaluar la veracidad de cada uno de sus componentes para determinar si toda la sentencia es verdadera o falsa. Para ello, se busca en el conjunto de axiomas la forma de establecer la veracidad de los predicados componentes. Un predicado componente se dice que es verdadero si se identifica con un axioma de la base de información. En la lógica de predicados, este proceso es algo complicado ya que las sentencias pueden tener términos variables. A los predicados que tienen variables por argumentos, se los denomina **patrones**.

Inferencia y Razonamiento

Inferir es concluir o decidir a partir de algo conocido o asumido; llegar a una conclusión. A su vez, razonar es pensar coherente y lógicamente; establecer inferencias o conclusiones a partir de hechos conocidos o asumidos.

El proceso de razonamiento, por lo tanto, involucra la realización de inferencias, a partir de hechos conocidos. Realizar inferencias significa derivar nuevos hechos a partir de un conjunto de hechos conocidos como verdaderos. La lógica de predicados proporciona un grupo de reglas sólidas, con las cuales se pueden realizar inferencias. Las principales **Reglas de Inferencia** son:

- **Modus ponens.-** Es la más importante, en los sistemas basados en conocimiento. Establece que:

  Si las sentencias p y $(p \rightarrow q)$ se conocen que son verdaderas,
  entonces se puede inferir que q también es verdadera.

- **Modus tolens.-** Esta regla establece que:

  Si la sentencia $(p \rightarrow q)$ es verdadera y q es falsa,
  entonces se puede inferir que p también es falsa.

- **Resolución.-** Utiliza refutación para comprobar una determinada sentencia. La refutación intenta crear una contradicción con la negación de la sentencia original, demostrando, por lo tanto, que la sentencia original es verdadera. La resolución es una técnica poderosa para probar teoremas en lógica y constituye la técnica básica de inferencia en **PROLOG**, un lenguaje que manipula en forma computacional la lógica de predicados. La regla de resolución, establece que:

  Si $(A \bigvee B)$ es verdadero y $(B \bigvee C)$ es verdadero,
  entonces $(A \bigvee C)$ también es verdadero.

En lógica de predicados, existen tres métodos básicos de razonamiento: deductivo, abductivo e inductivo.

- **Deducción.-** Es el razonamiento a partir de un principio conocido hacia un desconocido; de lo general, a lo específico, o de la premisa a la conclusión

lógica. La deducción realiza inferencias lógicamente correctas. Esto significa que la deducción a partir de premisas verdaderas, garantiza el resultado de conclusiones también verdaderas.

La deducción es el método más ampliamente comprendido, aceptado y reconocido de los tres indicados. Es la base tanto de la lógica proposicional, como de la lógica de predicados. A manera de ejemplo, el método deductivo, se puede expresar, utilizando lógica de predicados, como sigue:

$$\forall A, \forall B, \forall C, [mayor(A, B) \bigwedge mayor(B, C) \rightarrow mayor(A, C)]$$

- **Abducción.-** Es un método de razonamiento comúnmente utilizado para generar explicaciones. A diferencia de la inducción, la abducción no garantiza que se puedan lograr conclusiones verdaderas, por lo tanto no es un método sólido de inferencia. La forma que tiene la abducción es la siguiente:

  Si la sentencia $(A \rightarrow B)$ es verdadera y B es verdadera,
  entonces A es posiblemente verdadera.

  En abducción, se empieza por una conclusión y se procede a derivar las condiciones que podrían hacer a esta conclusión válida. En otras palabras, se trata de encontrar una explicación para la conclusión.

- **Inducción.-** Se define como el razonamiento a partir de hechos particulares o casos individuales, para llegar a una conclusión general. El método inductivo es la base de la investigación científica. La forma más común del método inductivo es la siguiente:

  Si se conoce que $P(a), P(b), \cdots, P(n)$ son verdaderos,
  entonces se puede concluir que $\forall X P(X)$ es también verdadero.

  La inducción es una forma de inferencia muy importante ya que el aprendizaje, la adquisición de conocimiento y el descubrimiento están basados en ella. Al igual que la abducción, la inducción no es un método sólido de inferencia.

El razonamiento deductivo es una forma **monotónica** de razonar que produce argumentos que preservan la verdad. En un sistema monotónico todos los axiomas

utilizados se conocen como verdaderos por sus propios méritos, o pueden ser derivados de otros hechos conocidos como verdaderos. Los axiomas no pueden cambiar, ya que una vez que se los conoce como verdaderos, siempre permanecen así y no pueden ser modificados o retractados. Esto significa que en el razonamiento monotónico el conjunto de axiomas continuamente crece en tamaño.

Otro aspecto del razonamiento monotónico es que si más de una inferencia lógica puede ser hecha a un tiempo específico y una de ellas se realiza, las inferencias que quedan serán todavía aplicables después que dicha inferencia haya sido hecha.

Ventajas y desventajas de la Lógica de Predicados

A continuación se presentan algunos aspectos característicos de la lógica de predicados y su implementación computacional, el lenguaje de programación PROLOG:

- **Manejo de incertidumbre.-** Una de las mayores desventajas de la lógica de predicados es que sólo dispone de dos niveles de veracidad: verdadero y falso. Esto se debe a que la deducción siempre garantiza que la inferencia es absolutamente verdadera. Sin embargo, en la vida real no todo es blanco y negro. En cierta forma el PROLOG ha logrado mitigar esta desventaja, permitiendo la inclusión de factores de certeza.

- **Razonamiento monotónico.-** La lógica de predicados al ser un formalismo de razonamiento monotónico, no resulta muy adecuada para ciertos dominios del mundo real, en los cuales las verdades pueden cambiar con el paso del tiempo. El PROLOG compensa esta deficiencia, proporcionando un mecanismo para remover los hechos de la base de datos. Por ejemplo, en TURBO PROLOG se tiene la cláusula **retractall**.

- **Programación declarativa.-** La lógica de predicados, tal como está diseñada en PROLOG, es un lenguaje de programación declarativo, en donde el programador sólo necesita preocuparse del conocimiento expresado en términos del operador de implicación y los axiomas. El mecanismo deductivo de la lógica de predicados llega a una respuesta (si esto es factible), utilizando un proceso exhaustivo de unificación y búsqueda. A pesar que la búsqueda exhaustiva puede ser apropiada en muchos problemas, también puede introducir ineficiencias durante la ejecución. Para lograr un cierto control en el proceso de búsqueda, PROLOG ofrece la operación de corte, CUT. Cuando no se utiliza el CUT, PROLOG se convierte en un lenguaje puramente declarativo.

1.3.3 Reglas de Producción

Son los sistemas más comúnmente utilizados. Su simplicidad y similitud con el razonamiento humano, han contribuido para su popularidad en diferentes dominios. Las reglas son un importante paradigma de representación del conocimiento.

Las reglas representan el conocimiento utilizando un formato **SI-ENTONCES** (**IF-THEN**), es decir tienen 2 partes:

- La parte **SI** (**IF**), es el antecedente, premisa, condición o situación;
- La parte **ENTONCES** (**THEN**), es el consecuente, conclusión, acción o respuesta.

Las reglas pueden ser utilizadas para expresar un amplio rango de asociaciones, por ejemplo:

SI está manejando un vehículo **Y** se aproxima una ambulancia,
ENTONCES baje la velocidad **Y** hágase a un lado para permitir el paso de la ambulancia.
SI su temperatura corporal es de 39 C,
ENTONCES tiene fiebre.
SI el drenaje del lavabo está tapado **Y** la llave de agua está abierta,
ENTONCES se puede inundar el piso.

Inferencia Basada en Reglas

Una declaración de que algo es verdadero o es un hecho conocido, es una **afirmación** (fact). El conjunto de afirmaciones se conoce a menudo con el nombre de **memoria de trabajo o base de afirmaciones**. De igual forma, al conjunto de reglas se lo denomina **base de reglas**.

Un sistema basado en reglas utiliza el *modus ponens* para manipular las afirmaciones y las reglas durante el proceso de inferencia. Mediante técnicas de búsqueda y procesos de unificación, los sistemas basados en reglas automatizan sus métodos de razonamiento y proporcionan una progresión lógica desde los datos iniciales, hasta las conclusiones deseadas. Esta progresión hace que se vayan conociendo nuevos hechos o descubriendo nuevas afirmaciones, a medida que va guiando hacia la solución del problema.

En consecuencia, el proceso de solución de un problema en los sistemas basados en reglas va realizando una serie de inferencias que crean un sendero entre la definición del problema y su solución. Las inferencias están concatenadas y se las realiza en forma progresiva, por lo que se lo que se dice que el proceso de solución origina una *cadena de inferencias*.

Los sistemas basados en reglas difieren de la representación basada en lógica en las siguientes características principales:

Son en general no-monotónicos, es decir hechos o afirmaciones derivadas, pueden ser retractados, en el momento en que dejen de ser verdaderos. Pueden aceptar incertidumbre en el proceso de razonamiento.

- Son en general no-monotónicos, es decir hechos o afirmaciones derivadas, pueden ser retractados, en el momento en que dejen de ser verdaderos.
- Pueden aceptar incertidumbre en el proceso de razonamiento.

El Proceso de Razonamiento

El proceso de razonamiento en un sistema basado en reglas es una progresión desde un conjunto inicial de afirmaciones y reglas hacia una solución, respuesta o conclusión. Como se llega a obtener el resultado, sin embargo, puede variar significativamente:

- Se puede partir considerando todos los datos conocidos y luego ir progresivamente avanzando hacia la solución. Este proceso se lo denomina guiado por los datos o de **encadenamiento progresivo** (*forward chainning*).
- Se puede seleccionar una posible solución y tratar de probar su validez buscando evidencia que la apoye. Este proceso se denomina guiado por el objetivo o de **encadenamiento regresivo** (*backward chainning*).

Razonamiento Progresivo

En el caso del razonamiento progresivo, se empieza a partir de un conjunto de datos colectados a través de observación y se evoluciona hacia una conclusión. Se chequea cada una de las reglas para ver si los datos observados satisfacen las premisas de alguna de las reglas. Si una regla es satisfecha, es ejecutada derivando nuevos hechos que pueden ser utilizados por otras reglas para derivar hechos adicionales. Este proceso de chequear reglas para ver si pueden ser satisfechas se denomina interpretación de reglas.

La interpretación de reglas es realizada por una máquina de inferencia en un sistema basado en conocimiento.

1. **Unificación (Matching).-** En este paso, en las reglas en la base de conocimientos se prueban los hechos conocidos al momento para ver cuáles son las que resulten satisfechas. Para decir que una regla ha sido satisfecha, se requiere que todas las premisas o antecedentes de la regla resuelvan a verdadero.

2. **Resolución de Conflictos.-** Es posible que en la fase de unificación resulten satisfechas varias reglas. La resolución de conflictos involucra la selección de la regla que tenga la más alta prioridad de entre el conjunto de reglas que han sido satisfechas.

3. **Ejecución.-** El último paso en la interpretación de reglas es la ejecución de la regla. La ejecución puede dar lugar a uno o dos resultados posibles: nuevo hecho (o hechos) pueden ser derivados y añadidos a la base de hechos, o una nueva regla (o reglas) pueden ser añadidas al conjunto de reglas (base de conocimiento) que el sistema considera para ejecución.

En esta forma, la ejecución de las reglas procede de una manera progresiva (hacia adelante) hacia los objetivos finales.

Un conjunto de aplicaciones adecuadas al razonamiento progresivo incluye supervisión y diagnóstico en sistemas de control de procesos en tiempo real, donde los datos están continuamente siendo adquiridos, modificados y actualizados. Estas aplicaciones tienen 2 importantes características:

1. Necesidad de respuesta rápida a los cambios en los datos de entrada.

2. Existencia de pocas relaciones predeterminadas entre los datos de entrada y las conclusiones derivadas.

Otro conjunto de aplicaciones adecuadas para el razonamiento progresivo está formado por: diseño, planeamiento y calendarización, donde ocurre la síntesis de nuevos hechos basados en las conclusiones de las reglas. En estas aplicaciones hay potencialmente muchas soluciones que pueden ser derivadas de los datos de entrada. Debido a que estas soluciones no pueden ser enumeradas, las reglas expresan conocimiento como patrones generales y las conexiones precisas entre estas reglas no pueden ser predeterminadas.

Razonamiento Regresivo

El mecanismo de inferencia, o interprete de reglas para el razonamiento regresivo, difiere significativamente del mecanismo de razonamiento progresivo. Si bien es cierto ambos procesos involucran el examen y aplicación de reglas, el razonamiento regresivo empieza con la conclusión deseada y decide si los hechos que existen pueden dar lugar a la obtención de un valor para esta conclusión. El razonamiento regresivo sigue un proceso muy similar a la búsqueda primero en profundidad.

El sistema empieza con un conjunto de hechos conocidos que típicamente está vacío. Se proporciona una lista ordenada de objetivos (o conclusiones), para las cuales el sistema trata de derivar valores. El proceso de razonamiento regresivo utiliza esta

lista de objetivos para coordinar su búsqueda a través de las reglas de la base de conocimientos.

Esta búsqueda consiste de los siguientes pasos:

1. Conformar una pila inicialmente compuesta por todos los objetivos prioritarios definidos en el sistema.

2. Considerar el primer objetivo de la pila. Determinar todas las reglas capaces de satisfacer este objetivo, es decir aquellas que mencionen al objetivo en su conclusión.

3. Para cada una de estas reglas examinar en turno sus antecedendentes:

   (a) Si todos los antecedentes de la regla son satisfechos (esto es, cada parámetro de la premisa tiene su valor especificado dentro de la base de datos), entonces ejecutar esta regla para derivar sus conclusiones. Debido a que se ha asignado un valor al objetivo actual, removerlo de la pila y retornar al paso (2).

   (b) Si alguna premisa de la regla no puede ser satisfecha, buscar reglas que permitan derivar el valor especificado para el parámetro utilizado en esta premisa.

   (c) Si en el paso (b) no se puede encontrar una regla para derivar el valor especificado para el parámetro actual, entonces preguntar al usuario por dicho valor y añadirlo a la base de datos. Si este valor satisface la premisa actual entonces continuar con la siguiente premisa de la regla. Si la premisa no es satisfecha, considerar la siguiente regla.

Si todas las reglas que pueden satisfacer el objetivo actual se han probado y todas no han podido derivar un valor, entonces este objetivo quedará indeterminado. Removerlo de la pila y retornar al paso (2). Si la pila está vacía parar y anunciar que se ha terminado el proceso.

El razonamiento regresivo es mucho más adecuado para aplicaciones que tienen mucho mayor número de entradas, que de soluciones posibles. La habilidad de la lógica regresiva para trazar desde las pocas conclusiones hacia las múltiples entradas la hace más eficiente que el encadenamiento progresivo.

Una excelente aplicación para el razonamiento regresivo es el diagnóstico, donde el usuario dialoga directamente con el sistema basado en conocimiento y proporciona los datos a través del teclado. Problemas de clasificación también son adecuados para ser resuelto mediante el razonamiento regresivo.

Arquitecturas basadas en Reglas

El tipo de conocimiento descrito con sistemas basados en reglas varían significativamente en complejidad.

Algunas veces las conclusiones derivadas de las reglas pueden ser hechos que se identifican en forma exacta con las premisas de otras reglas. En estos casos, se puede visualizar una base de conocimientos como una red de reglas y hechos interconectados.

En otros casos, las conclusiones derivadas pueden ser más generales. Como resultado, la visualización de la base de conocimiento como una red, no es posible aplicarla. En lugar de esto, nos vemos forzados a pensar que las conclusiones derivadas de las reglas son una colección de hechos que podrían o no unificarse o identificarse con los varios patrones descritos por las premisas de otras reglas.

Esto da como resultado dos tipos de estructuras y organizaciones al conocimiento contenido dentro de un sistema basado en reglas: *redes de inferencia y sistemas de unificación de patrones*.

Cabe señalar que ambas arquitecturas pueden trabajar con encadenamiento progresivo o regresivo. Sin embargo, tradicionalmente se ha utilizado el proceso de razonamiento regresivo en redes de inferencia y el proceso de razonamiento progresivo en sistemas de unificación de patrones.

Redes de Inferencia

Una red de inferencia puede ser representada como un gráfico en el que los nodos representan parámetros que son los hechos obtenidos como datos o derivados de otros datos. Cada parámetro es una declaración acerca de algún aspecto del problema bajo análisis y puede servir como un antecedente o consecuente de una regla. Estas declaraciones pueden copar un rango que va desde la conclusión final de un sistema, hasta hechos simples, observados o derivados. Cada uno de estos parámetros puede tener uno o más valores asociados, donde cada valor tiene una medida correspondiente de incertidumbre que representa cuan creíble es el valor particular de un parámetro.

Las reglas en el sistema están representadas dentro del gráfico por las interconexiones entre los varios nodos. Este conocimiento es utilizado por el proceso de inferencia para propagar resultados a través de la red.

Nótese que todas las interconexiones entre los varios nodos de la red de inferencia son conocidas previa a la ejecución del sistema. Esto trae como consecuencia la minimización del proceso de búsqueda de hechos que se identifiquen con las premisas. Adicionalmente, simplifican la implementación del mecanismo de inferencia y el manejo de las facilidades de explicación.

Las redes de inferencia son muy útiles para dominios donde el número de diferentes soluciones alternativas es limitado. Por ejemplo, la clasificación de elementos en las ciencias naturales y problemas de diagnóstico. Una red de inferencia es fácil de

implementar, pero es menos poderosa ya que se debe conocer de antemano todas las relaciones entre reglas y hechos.

Sistemas comerciales de desarrollo, basados en esta arquitectura son los siguientes: Personal Consultant, EXSYS, y VP-Expert.

Sistemas de Unificación de Patrones

Estos sistemas utilizan procesos de búsqueda extensivos para unificar y ejecutar las reglas. Típicamente usan complejas implementaciones de asociación de patrones para asignar valores a variables, condicionar los valores permisibles para ser asociados a una premisa y para determinar las reglas a ejecutar. Las relaciones entre las reglas y los hechos son formadas durante la ejecución, basados en los patrones que se identifican con los hechos.

La unificación de patrones es una idea importante y poderosa en razonamiento automatizado que fue utilizada por primera vez en el lenguaje PROLOG.

Los sistemas basados en reglas que utilizan la unificación de patrones son extremadamente flexibles y poderosos. Son más aplicables a dominios en los que las posibles soluciones son, ya sea ilimitadas o muy grandes en número, tales como diseño, planeamiento y síntesis. Sin embargo, el uso de procesos de búsqueda para encontrar reglas aplicables en los sistemas de unificación de patrones los puede volver ineficientes en implementaciones grandes.

Comercialmente existen varios sistemas de desarrollo basados en esta arquitectura: XCON, OPS-5, ART, CLIPS, y KEE.

Desventajas de las Reglas de Producción

Algunos problemas existen en los sistemas basados en reglas. Estos problemas caen dentro de una de las siguientes categorías: encadenamiento infinito; incorporación de conocimiento nuevo contradictorio, y; modificación de reglas existentes.

Desventajas adicionales pueden ser: ineficiencia (necesidad de modularizar o de introducir metarreglas), opacidad (dificultad de establecer relaciones), adaptación al dominio (rápido crecimiento del número de reglas).

El conocimiento acerca de las reglas de producción se denomina **METARREGLA**. Las **metarreglas** facilitan y aceleran la búsqueda de soluciones.

Ventajas de las Reglas de Producción

A pesar de las desventajas anotadas, los sistemas basados en reglas han permanecido como los esquemas más comúnmente utilizados para la representación del conocimiento. Como ventajas significativas se pueden mencionar las siguientes: modularidad, uniformidad y naturalidad para expresar el conocimiento.

1.3.4 Redes Asociativas

Las **redes semánticas**, fueron originalmente desarrolladas para representar el significado o semántica de oraciones en inglés, en términos de objetos y relaciones. Actualmente, el término **redes asociativas** es más ampliamente utilizado para describirlas ya que no sólo se las usa para representar relaciones semánticas, sino también para representar asociaciones físicas o causales entre varios conceptos u objetos.

Las redes asociativas se caracterizan por representar el conocimiento en forma gráfica. Agrupan una porción de conocimiento en dos partes: objetos y relaciones entre objetos. Los objetos se denominan también nodos (elementos del conocimiento) y las relaciones entre nodos se denominan enlaces o arcos. Cada nodo y cada enlace en una red semántica, deben estar asociados con objetos descriptivos.

Son muy apropiadas para representar conocimiento de naturaleza jerárquica. Su concepción se basa en la asociación de conocimientos que realiza la memoria humana. Las principales aplicaciones son: comprensión de lenguaje natural, bases de datos deductivas, visión por computadora, sistemas de aprendizaje.

Ventajas de las Redes Asociativas

Las redes asociativas tienen dos ventajas sobre los sistemas basados en reglas y sobre los basados en lógica:

1. Permiten la declaración de importantes asociaciones, en forma explícita y sucinta.

2. Debido a que los nodos relacionados están directamente conectados, y no se expresan las relaciones en una gran base de datos, el tiempo que toma el proceso de búsqueda por hechos particulares puede ser significativamente reducido.

Desventajas de las Redes Asociativas

Entre las desventajas de las redes asociativas, se pueden mencionar:

1. No existe una interpretación normalizada para el conocimiento expresado por la red. La interpretación de la red depende exclusivamente de los programas que manipulan la misma.

2. La dificultad de interpretación a menudo puede derivar en inferencias inválidas del conocimiento contenido en la red.

3. La exploración de una red asociativa puede derivar en una explosión combinatoria del número de relaciones que deben ser examinadas para comprobar una relación.

1.3.5 Representación mediante Plantillas

Una plantilla *(frame)* es una estructura de datos apropiada para representar una situación estereotípica. Las plantillas organizan el conocimiento en objetos y eventos que resultan apropiados para situaciones específicas. Evidencia psicológica sugiere que la gente utiliza grandes plantillas para codificar el conocimiento de experiencias pasadas, o conocimiento acerca de cosas que se encuentran comúnmente, para analizar y explicar una situación nueva en su cotidiana actividad cognoscitiva.

Una *plantilla* representa un objeto o situación describiendo la colección de atributos que posee. Están formadas por un nombre y por una serie de campos de información o *ranuras (slots)*. Cada ranura puede contener uno o más *enlaces (facets)*. Cada *enlace* tiene un valor asociado. Varios enlaces pueden ser definidos para cada *ranura*, por ejemplo:

- *Rango.*- El conjunto de posibles valores para la ranura.
- *Valor.*- El valor de la ranura.
- *Default.*- El valor a ser asumido si no se especifica alguno.

Además los enlaces pueden ser procedimientos que residen en la base de datos y están aguardando para ser utilizados cuando se los necesite. Entre los más comunes se pueden mencionar:

- *Si-Necesitado.*- Procedimiento(s) para determinar el valor actual de una ranura.
- *Si-Agregado.*- Procedimiento(s) a ejecutarse cuando un valor es especificado para una ranura.
- *Si-Modificado.*- Procedimiento(s) a ejecutarse si el valor de una ranura es cambiado.

A estos procedimientos también se los denomina *demons* y representan un concepto poderoso en las plantillas, esto es, la habilidad de combinar conocimiento procedimental dentro de la estructura de conocimiento declarativo de la plantilla. Esto sugiere que una plantilla puede ser un medio poderoso de representación del conocimiento, especialmente si se la incorpora en una red de plantillas.

En la siguiente figura se muestra una representación abstracta de una plantilla.

Se pueden establecer ciertas similitudes entre un sistema basado en plantillas y un sistema de bases de datos. Aparentemente los dos representan "datos"(a través de las ranuras de una plantilla y de los campos de una tabla de datos), sin embargo las plantillas representan en realidad conocimiento, mientras que las bases de datos representan sólo datos. La investigación que se realiza actualmente en bases de datos

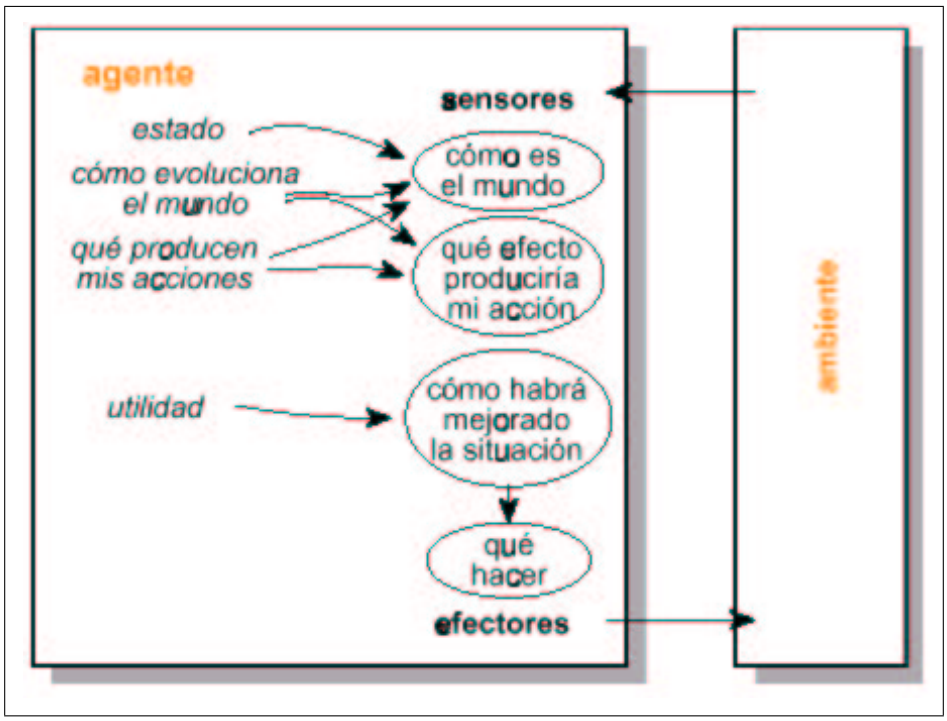

Figura 1.11: Representación abstracta de una plantilla.

está examinando la posibilidad de aplicarlas a la representación del conocimiento, incorporando herencia y demons (**Bases de Datos Inteligentes**), similar a lo que se tiene en sistemas basados en conocimiento.

Ventajas de las Plantillas

Las ventajas que se pueden establecer para los sistemas basados en plantillas son las siguientes:

1. Facilidad de proceso guiado por las expectativas. Un sistema basado en plantillas, mediante los *demons* es capaz de especificar acciones que deben tener lugar cuando ciertas condiciones se han cumplido durante el procesamiento de la información.

2. El conocimiento que posee un sistema basado en plantillas es significativamente más estructurado y organizado que el conocimiento dentro de una red asociativa.

3. Las plantillas pueden ser estructuradas de tal forma que sean capaces de determinar su propia aplicabilidad en determinadas situaciones. En el caso de que una plantilla en particular no sea aplicable, puede sugerir otras plantillas que pueden ser apropiadas para la situación.

4. Se puede fácilmente almacenar en las ranuras valores dinámicos de variables, durante la ejecución de un sistema basado en conocimiento. Esto puede ser particularmente útil para aplicaciones de simulación, planeamiento, diagnóstico de problemas o interfaces para bases de datos.

Desventajas de las Plantillas

Las principales desventajas que se pueden establecer para la representación del conocimiento mediante plantillas, son:

1. Dificultad de representar objetos que se alejen considerablemente de estereotipos.

2. No tiene la posibilidad de acomodarse a situaciones u objetos nuevos.

3. Dificultad para describir conocimiento heurístico que es mucho más fácilmente representado mediante reglas.

1.3.6 Representación mediante Objetos

Los objetos, son similares a las plantillas. Ambos sirven para agrupar conocimiento asociado, soportan herencia, abstracción y el concepto de procedimientos agregados. La diferencia radica en los siguiente:

1. En las plantillas, a los programas y a los datos se los trata como dos entidades relacionadas separadas. En cambio en los objetos se crea una fuerte unidad entre los procedimientos *(métodos)* y los datos.

2. Los *demons* de las plantillas sirven sólo para computar valores para las diversas ranuras o para mantener la integridad de la base de conocimientos cada vez que una acción de alguna plantilla, afecta a otra. En cambio, los métodos utilizados por los objetos son más universales ya que proporcionan cualquier tipo general de computación requerida y además soportan encapsulamiento y polimorfismo.

Un objeto es definido como una colección de información representando una entidad del mundo real y una descripción de cómo debe ser manipulada esta información, esto es los métodos. Es decir, un objeto tiene un nombre, una caracterización de clase, varios atributos distintivos y un conjunto de operaciones. La relación entre los objetos viene definida por los mensajes. Cuando un objeto recibe un mensaje válido, responde con una acción apropiada, retornando un resultado.

Ventajas de la Representación mediante Objetos

Los objetos, como forma de representación del conocimiento ofrece las siguientes ventajas:

1. Poder de abstracción.
2. Encapsulamiento o capacidad de esconder información.
3. Herencia, es decir pueden recibir características o propiedades de sus ancestros.
4. Polimorfismo, que permite crear una interfaz común para todos los diversos objetos utilizados dentro del dominio.
5. Posibilidad de reutilización del código.
6. Mayor facilidad para poder trabajar eficientemente con sistemas grandes.

Desventajas de la Representación mediante Objetos

Las desventajas son similares a las que se indicaron para las plantillas:

1. Dificultades para manejar objetos que se alejan demasiado de la norma.
2. Dificultades para manejar situaciones que han sido encontradas previamente.

1.4 Lenguajes de programación y modelado

1.4.1 Prolog

La Programación Lógica y uno de sus representantes, PROLOG, hace un enfoque declarativo para escribir programas para el computador. Los programas lógicos pueden ser entendidos y estudiados usando dos conceptos abstractos: verdad y deducción lógica. Uno puede preguntar si un axioma en un programa es verdad, bajo alguna interpretación de los símbolos del programa, o si una instrucción lógica es una consecuencia del programa. Esas preguntas pueden ser respondidas independientemente de cualquier mecanismo de ejecución concreto.

Por el contrario, Prolog es un lenguaje de programación con un significado operacional preciso, que toma prestado sus conceptos básicos de la programación lógica. Los programas Prolog son instrucciones para ser ejecutados sobre el computador. Esas instrucciones casi siempre son leídas como instrucciones lógicas y, lo más importante, el resultado de una computación de un programa Prolog es una consecuencia lógica de los axiomas en éste.

Hay que notar, que una programación de Prolog efectiva requiere una comprensión de la teoría de programación lógica.

Breve historia de Prolog

El comienzo de la programación lógica puede ser atribuido a Kowalski y Colmerauer. Kowalski formuló la interpretación procedimental de la lógica de cláusulas de Horn y mostró que el axioma *A Si B* puede ser leído como un procedimiento de un lenguaje de programación recursivo, donde A es la cabeza del procedimiento y B su cuerpo. Al mismo tiempo, a principios de 1970, Colmerauer y su grupo en la Universidad De Marseille-Aix desarrolló un probador de teorema especializado, el cual ellos usaron para implementar sistemas de procesamiento natural. El probador de teorema, lo llamaron PROLOG (for Programation et Logique o Programming in Logic), basado en la interpretación procedimental de Kowalski.

El primer interpretador de Prolog no fue tan rápido como los sistemas Lisp pero esto cambió a mediados de 1970 cuando David H.D. Warren y sus colegas desarrollaron una implementación eficiente de Prolog. El compilador, el cual fue casi completamente escrito en Prolog, tradujo cláusulas de Prolog hacia instrucciones de máquina abstracta que es ahora conocida como Warren Abstract Machine (WAM). Sin embargo, la comunidad de Inteligencia Artificial y Ciencia De La Computación de Western estuvo aún ignorante e indiferente a la programación lógica.

Esto cambió un poco con el anuncio del Proyecto de Quinta Generación Japonesa el cual claramente afirmó el rol importante de la programación lógica en la próxima generación de sistemas computarizados. No sólo los investigadores sino también el público en general comenzó a deletrear la palabra PROLOG. Desde esa vez, Prolog está desarrollando para nuevas alturas. Actualmente, una de las extensiones más prominentes es la programación lógica con restricciones (Contraint Logic Programming-CLP).

Comandos para el uso del sistema Prolog

Un programa Prolog es un conjunto de procedimientos donde su orden es indiferente, cada procedimiento consiste de una o más cláusulas (pero, el orden de las cláusulas es importante). Hay dos tipos de cláusulas: Hechos y Reglas. El programa es almacenado en una base de datos Prolog, que usualmente es cargado usando el comando *"consult"* de la siguiente forma:

$$?- \mathit{consult}(\texttt{'archivo'}).$$

El comando *consult* agrega las cláusulas y hechos desde el archivo texto especificado a las cláusulas y hechos ya almacenados en la base de datos. Así se puede cargar más programas dentro de la base de datos a la vez pero hay que ser cuidadoso

si los programas no usan los procedimientos con el mismo nombre. De otra manera, debido a la acumulación de cláusulas, esos procedimientos podrían comportarse incorrectamente.

También se puede usar el comando *reconsult* para cargar un programa.

$$? - reconsult('\texttt{archivo}').$$

Este comando se comporta igual al comando *consult* (agrega procedimientos dentro de la base de datos) pero si hay un procedimiento en la base de datos con el mismo nombre como cualquier procedimiento en el archivo reconsultado, entonces el primer procedimiento es reemplazado por la nueva definición. Usualmente se usa el comando *reconsult* para cambiar un programa en la base de datos durante la depuración.

El programa Prolog es comenzado llamando algún procedimiento del programa de la siguiente forma:

$$? - \texttt{procedimiento}(\texttt{parametros}).$$

Hay que acotar, que cuando se llama un procedimiento, cuando se consulta o reconsulta el archivo. Este procedimiento es llamado realizando una pregunta a Prolog.

Ahora si presentamos un programa de prolog simple, para expresar la propiedad de ser un hombre, mujer o pariente; siguiendo hechos de Prolog, sería de la siguiente forma:

$$\begin{array}{l} hombre(adam). \\ hombre(peter). \\ hombre(paul). \\ mujer(marry). \\ mujer(eve). \\ pariente(adam, peter). \\ pariente(eve, peter). \\ pariente(adam, paul). \\ pariente(marry, paul). \end{array}$$

Hasta ahora, hemos agregado sólo hechos a nuestro programa pero el poder real de Prolog está en las reglas. Mientras que los hechos afirman la relación explícitamente, las reglas definen la relación en una forma más general. Cada regla tiene su cabeza(nombre de la relación definida), y su cuerpo (una definición real de la relación). Las siguientes reglas definen las relaciones de ser un padre y ser una madre usando

las relaciones definidas previamente de ser un hombre o mujer y ser un pariente.

$$padre(F, C) : -hombre(F), pariente(F, C).$$
$$madre(M, C) : -mujer(M), pariente(M, C).$$

Si observamos en las reglas anteriores usamos variables las cuales comienzan con mayúscula, para expresar la característica que cada hombre el cual es un pariente de cualquier niño es también su padre. Si algún parámetro de la relación no es importante podemos usar variables anónimas (denotadas _) como en las definiciones siguientes:

$$es\_padre(F) : -padre(F, \_).$$
$$es\_madre(M) : -madre(M, \_).$$

Ahora que ya tenemos una idea de que es un hecho, una regla y una variable, un programa en prolog lo podemos ejecutar, haciendo preguntas como estas:

$$? - padre(X, paul).$$

Lo cual expresa: ¿Quién es padre de Paul? La respuesta es X=*adam*, naturalmente.

El cuerpo de la regla puede también usar la relación que está siendo definida. Esta característica es llamada recursión y las siguientes reglas muestran su uso típico-.

$$descendiente(D, A) : -pariente(A, D).$$
$$descendiente(D, A) : -pariente(P, D), descendiente(P, A).$$

Uno puede usar la característica de Prolog de variables de entrada y salida no determinadas y fácilmente definir la relación ancestro.

$$ancestro(A, D) : -descendiente(D, A).$$

La estructura de datos básica en Prolog es el *término* el cual es expresado *nombre(argumentos...)*. Si el número de argumentos es cero entonces estamos hablando de un *átomo*. Un tipo especial de átomo es el *número*.

Existe también estructuras de datos ampliamente usada y construida en Prolog, que se les denomina *listas*, son todavía un término, ejemplo, *[1,2,3]* y es equivalente a *'.'(1,'.'(2,'.'(3,nulo)))*. Las siguientes funciones permiten accesar los elementos de

la lista.

$$cabeza(C, [C|\_]).$$
$$cola(L, [\_|L]).$$

Es fácil accesar el primer elemento de la lista, es decir, la cabeza. Sin embargo, encontrar el último elemento es un proceso que consume tiempo ya que uno tiene que recorrer la lista completa para encontrarlo. Pero, se pueden realizar "procedimientos"para encontrar el primero/último elemento de la lista o para generar una lista con el primero/último elemento dado. Un procedimiento podría ser algo como esto:

$$primero(P, [P|\_]).$$
$$ultimo(U, [U]).$$
$$ultimo(U, [C|L]) : -ultimo(U, L).$$

En fin, son muchas cosas que hay que conocer de Prolog y que se pueden realizar con este lenguaje de programación lógica, aquí hemos realizado una descripción muy general de este lenguaje, pero si se quiere profundizar más sobre Prolog, en internet existen una diversidad de manuales y tutoriales, que nos explican de una manera muy sencilla, las bondades que ofrece Prolog.

1.4.2 JPL

El JPL, es un conjunto de clases de Java y funciones en C que proveen una interface entre Java y Prolog. JPL usa la Interface Nativa de Java (JNI) para conectarse a una máquina Prolog a través de la interface de Lenguaje Exterior (FLI). JPL no es una implementación de Java Puro de Prolog, en cambio, éste hace uso extensivo de implementaciones nativas de Prolog sobre plataformas soportadas. La versión actual de JPL sólo trabaja con SWI-Prolog.

Actualmente, JPL sólo soporta empotrar una máquina prolog dentro de la máquina virtual Java. JPL está diseñado en dos capas, una interface de bajo nivel a la FLI de Prolog (para programadores de C) y una interface en Java de Alto nivel para el programador de Java.

1.4.3 Java

Es un lenguaje de programación que está diseñado para cumplir los requisitos de entrega de contenidos interactivos mediante la utilización de applet insertadas en las páginas HTML. Dado que el éxito de Java depende de la red mundial(*World Wide Web*), las clases básicas de manipulación de cadenas están adecuadas para el tipo de

procesamiento de texto que forma parte de cualquier programa basado en HTML.

Los diferentes tipos de sistemas con los cuales los usuarios exploran la Web (PC, Macs, estaciones de trabajo, y otros) utilizan un tipo de CPU (procesador) diferente. Así por ejemplo cuando se crean programas en lenguajes como C++, el compilador de Java genera un código intermedio, llamado de máquina virtual, que no es para un procesador específico, permitiendo independencia de plataforma.

Java es un lenguaje de programación de alto nivel que ofrece la potencia del diseño orientado a objetos con una sintaxis simple y familiar, en un entorno robusto y agradable de utilizar. Posee un conjunto de clases de objetos que proporcionan al programador abstracciones claras para muchas funciones de sistema habituales, como la gestión de ventanas, de red y entrada y salida. La clave de estas clases es que proporcionan una abstracción multi-plataforma para una gran variedad de interfaces.

Las características principales que nos ofrece Java respecto a cualquier otro lenguaje de programación, son:

- **Es simple:** Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos. C++ es un lenguaje que adolece de falta de seguridad, pero C y C++ son lenguajes más difundidos, por ello Java se diseñó para ser parecido a C++ y así facilitar un rápido y fácil aprendizaje.

  Java elimina muchas de las características de otros lenguajes como C++, para mantener reducidas las especificaciones del lenguaje y añadir características muy útiles como el *garbage collector* (reciclador de memoria dinámica). No es necesario preocuparse de liberar memoria, el reciclador se encarga de ello y como es un *thread* de baja prioridad, cuando entra en acción, permite liberar bloques de memoria muy grandes, lo que reduce la fragmentación de la memoria.

- **Es orientado a objetos:** Java implementa la tecnología básica de C++ con algunas mejoras y elimina algunas cosas para mantener el objetivo de la simplicidad del lenguaje. Java trabaja con sus datos como objetos y con interfaces a esos objetos. Soporta las tres características propias del paradigma de la orientación a objetos: encapsulamiento, herencia y polimorfismo. Las plantillas de objetos son llamadas, como en C++, clases y sus copias, instancias. Estas instancias, como en C++, necesitan ser construidas y destruidas en espacios de memoria.

- **Es distribuido:** Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como HTTP y FTP. Esto permite a los programadores acceder a la información a través de la red con tanta facilidad como a los ficheros locales.

La verdad es que Java en sí no es distribuido, sino que proporciona las librerías y herramientas para que los programas puedan ser distribuidos, es decir, que se corran en varias máquinas interactuando.

- **Es robusto:** Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador de la liberación o corrupción de memoria. También implementa los arrays auténticos, en vez de listas enlazadas de punteros, con comprobación de límites, para evitar la posibilidad de sobreescribir o corromper memoria resultado de punteros que señalan a zonas equivocadas. Estas características reducen drásticamente el tiempo de desarrollo de aplicaciones en Java.

- **Es de arquitectura neutral:** Para establecer Java como parte integral de la red, el compilador Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (run-time) puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado. Actualmente existen sistemas run-time para Solaris 2.x, SunOs 4.1.x, Windows 95, Windows NT, Linux, Irix, Aix, Mac, Apple y probablemente haya grupos de desarrollo trabajando en el porting a otras plataformas.

- **Es seguro:** El código Java pasa muchos tests antes de ejecutarse en una máquina. El código se pasa a través de un verificador de byte-codes que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal -código que falsea punteros, viola derechos de acceso sobre objetos o intenta cambiar el tipo o clase de un objeto.las aplicaciones de Java resultan extremadamente seguras, ya que no acceden a zonas delicadas de memoria o de sistema, con lo cual evitan la interacción de ciertos virus. Java no posee una semántica específica para modificar la pila de programa, la memoria libre o utilizar objetos y métodos de un programa sin los privilegios del kernel del sistema operativo. Además, para evitar modificaciones por parte de los crackers de la red, implementa un método ultraseguro de autentificación por clave pública. El Cargador de Clases puede verificar una firma digital antes de realizar una instancia de un objeto. Por tanto, ningún objeto se crea y almacena en memoria, sin que se validen los privilegios de acceso. Es decir, la seguridad se integra en el momento de compilación, con el nivel de detalle y de privilegio que sea necesario.

- **Es portable:** Más allá de la portabilidad básica por ser de arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros y además, enteros de 32 bits en complemento a 2. Además, Java construye sus interfaces de usuario a través de un sistema abstracto de ventanas de forma que las ventanas puedan ser implantadas en entornos Unix, Pc o Mac.

- **Es interpretado:** Para conseguir ser un lenguaje independiente del sistema operativo y del procesador que incorpore la máquina utilizada, es tanto interpretado como compilado. Y esto no es ningún contrasentido, el código fuente escrito con cualquier editor se compila generando el byte-code. Este código intermedio es de muy bajo nivel, pero sin alcanzar las instrucciones máquina propias de cada plataforma y no tiene nada que ver con el p-code de Visual Basic. El byte-code corresponde al 80% de las instrucciones de la aplicación. Ese mismo código es el que se puede ejecutar sobre cualquier plataforma. Para ello hace falta el run-time, que sí es completamente dependiente de la máquina y del sistema operativo, que interpreta dinamicamente el byte-code y añade el 20% de instrucciones que faltaban para su ejecución. Con este sistema es fácil crear aplicaciones multiplataforma, pero para ejecutarlas es necesario que exista el run-time correspondiente al sistema operativo utilizado.

- **Es multihebra:** Al ser multihebra (multithread), Java permite muchas actividades simultáneas en un programa. Los threads (a veces llamados, procesos ligeros), son básicamente pequeños procesos o piezas independientes de un gran proceso. Al estar los threads construidos en el lenguaje, son más fáciles de usar y más robustos que sus homólogos en C o C++.

1.4.4 UML

Los proyectos de Software son complejos, y la estrategia primaria para superar la complejidad es la descomposición, es decir, dividir el problema en unidades manejables. Antes del advenimiento del análisis y diseño orientado a objetos, el método usual de descomponer el problema era el análisis y diseño estructurado cuya dimensión de descomposición es fundamentalmente por funciones o procesos.

La esencia del análisis y el diseño orientado a objetos consiste en situar el dominio del problema y su solución lógica dentro de la perspectiva de los objetos. Se procura ante todo identificar y describir los objetos o conceptos lógicos del software, que finalmente será implementado en un lenguaje de programación orientado a objetos donde se define para cada objeto sus atributos y sus métodos.

El UML (*Unified Modeling Language*) o Lenguaje Unificado de Modelado, se define como un lenguaje de notación gráfica que permite especificar, visualizar, construir y

documentar los objetos de los sistemas de software, mediante notación esquemática en su mayor parte, bajo los conceptos de orientación a objetos.

UML Nació en 1994 para combinar dos famosos métodos: el de Grady Booch, denominado Boochy el método de Jim Rumbaugh, OMT (*Oriented Modeling Technique*). Mas tarde se les unió Ivar Jacobson, creador de OOSE (*Object-Oriented Software Engineering*). UML surge como petición de OMG (*Object Management Group*) para fijar estándares en la industria, definiendo un lenguaje y una notación de construcción de modelos. Luego de 1997, UML recibió la aprobación de la industria, al representar métodos muy difundidos de la primera generación de análisis y diseño orientado a objetos. [Rational:1991]

El primer paso al utilizar UML desde el punto de vista del método de análisis y diseño orientado a objetos, se denomina Análisis de Requerimientos, en el cual los procesos y las necesidades se descubren, la meta primaria es identificarlas y documentarlas en forma clara para el cliente y los miembros del equipo de desarrollo. Los requerimientos deben ser definidos de manera inequívoca, de modo de detectar los riesgos.

Los conceptos básicos de UML se agrupan por tipos de diagramas:

1. **Diagrama de Casos de Uso**

   Los Casos de Uso son descripciones de los procesos de un sistema. Se describe la secuencia de eventos de un actor o agente externo, que de alguna manera participa en el sistema. Por lo regular estimula al sistema con eventos de entrada o recibe algo de él.

   Cada Caso de Uso es una operación completa desarrollada por los actores y el sistema. El conjunto de Casos de Uso representa la totalidad de operaciones del sistema.

2. **Diagramas de Clases**

   Describe las especificaciones de los objetos de software y de las interfaces en una aplicación. Normalmente contiene:

   - Clases, asociaciones y atributos
   - Interfaces, con sus operaciones y constantes
   - Métodos
   - Información sobre los tipos de atributo
   - Navegabilidad
   - Dependencias

Aquí se presenta el conjunto de clases y objetos importantes que forman parte de un sistema, además de sus relaciones. Muestra de una manera estática la estructura de información del sistema y la visibilidad que tiene cada una de las clases.

3. **Diagramas de Interacción**

   Son modelos que describen cómo los grupos de objetos colaboran en algunos ambientes. Captura el comportamiento de un único Caso de Uso.

4. **Diagramas de secuencia**

   Muestran la interacción de un conjunto de objetos en una aplicación a través del tiempo. Puede dar detalle a los Casos de Uso. Un objeto se representa como una línea vertical punteada llamada línea de vida con un rectángulo de encabezado y con rectángulos a través de la línea principal que denotan la activación o el período de tiempo en el cual el objeto se encuentra desarrollando alguna operación. Los Diagramas de Secuencia, describen gráficamente las interacciones del actor y de las operaciones a que dan origen, mostrándolas en determinado escenario de un Caso de uso.

5. **Diagramas de Colaboración**

   Representan interacción entre objetos o la relación entre ellos, y la secuencia de los mensajes de las iteraciones que están indicadas por un número.

   Una vez identificados los procesos y quienes intervienen, se determina la manera de cumplir los procesos o la asignación de responsabilidades, distribuyendo las funciones y responsabilidades entre varios objetos del software en la aplicación. Del mismo modo que se asignan a los papeles de quienes intervienen.

   La descripción de la asignación de las responsabilidades y las interacciones de objetos a menudo son expresadas gráficamente con diagramas de diseño de clases y con diagramas de colaboración, mostrando la definición de las clases y el flujo de mensajes entre los objetos del software.

6. **Diagramas de Estados**

   Muestra el conjunto de estados por los cuales pasa un objeto durante su vida en una aplicación, junto con los cambios que permiten pasar de un estado a otro.

   Una vez concluidos los diagramas de clases de diseño, y destinados al ciclo de desarrollo actual en la aplicación, se disponen de suficientes detalles para generar un código que utilizaremos. Los artefactos creados en la parte de diseño, los diagramas de colaboración, y los de las clases de diseño, servirán de entrada en el proceso de generación de código.

Si se quiere reducir el riesgo y aumentar la probabilidad de conseguir una aplicación adecuada, el desarrollo debería basarse en un buen análisis, modelado y diseño antes de iniciar la codificación. Las herramientas modernas de desarrollo ofrecen un excelente ambiente para examinar rápidamente métodos alternos.

La creación de código en un lenguaje orientado a objetos, no forma parte del análisis ni del diseño orientado a objetos. Los metodólogos seguirán definiendo técnicas, modelos, y procesos de desarrollo para la creación eficaz de sistemas de software; sólo que ahora podrán hacerlo en un lenguaje común: UML.

7. **Diagramas de Actividades**

Es un caso especial de un Diagrama de Estado en el cual casi todos los estados son estados de acción y casi todas las transiciones son enviadas al terminar la acción ejecutada en el estado anterior. Generalmente modelan los pasos de un algoritmo y pueden dar los detalles de un Caso de Uso, un objeto o un mensaje de éste. Sirven para representar transiciones internas.

Capítulo 2

Desarrollo del prototipo

En este capítulo nos concentramos en la especificación de los requerimientos y construcción del prototipo del motor de inferencia, el cual permite incorporar al simulador GALATEA, la incorporación de un agente.

Utilizando el método interactivo basado en casos de usos, hablaremos sobre los requerimientos funcionales, la planificación a seguir en las diferentes fases de construcción realizadas, y finalmente realizando el perfeccionamiento del prototipo se realizará las pruebas a través de casos de uso con el sistema para su aceptación.

2.1 Análisis y especificación de requerimientos

2.1.1 Determinación de los requerimientos de información

Los resultados arrojados por el motor de inferencia debe ser mostrados de la siguiente manera:

- La salida generada por el motor de inferencia, es escrita en un archivo *salidap.data*, y es impresa en consola
- El agente muestra en el caso de consultar un hecho, *"true"* si es verdadero o *"false"* si es falso.
- Cuando se consulta una regla, si la regla es verdadera muestra una solución, desplegando en pantalla las soluciones de las variables consultadas en el mismo orden como fueron introducidas en la entrada y en el archivo las soluciones de las variables se van escribiendo cada una en una línea diferente. En el caso de que la regla posea variables repetidas despliega únicamente su resultado en el mismo orden que aparece por primera vez.

- Si la regla no se logra satisfacer, el programa despliega *"false"*

2.1.2 Determinación de los requerimientos de funcionales

- Su diseño e implantación debe seguir las pautas del Proyecto GALATEA [Uzcátegui:2002]
- El motor de inferencia debe funcionar de manera independiente, poder recibir cualquier tipo de meta y proporcionar respuesta de manera inmediata.
- Se debe adaptar a cualquier tipo de base de conocimiento que se pueda consultar, haciendo que su arquitectura sea lo más genérica y robusta posible.
- Debe ser eficiente a la hora de arrojar sus repuestas, el tiempo de respuestas debe ser mínimo, el tal sentido el procesamiento de una regla por el motor de inferencia debe ser mínimo (su algoritmo de procesamiento debe ser óptimo), dependiendo únicamente del tiempo de procesamiento del procesador del computador donde se corra el motor de inferencia.
- La salida del motor de inferencia debe ser lo más genérica posible, y de esta manera se pueda acoplar de manera eficiente a cualquier programa donde se quiera utilizar el agente.
- El motor de inferencia debe ser de fácil manipulación, en tal sentido las clases que lo componen debe tener métodos independientes, permitiendo utilizar todas las bondades que brinda la orientación por objetos para su reutilización.
- El motor de inferencia que posea el agente debe ser versátil a la hora de su utilización, pudiendo implementar sus métodos dentro de nuestros propios programas, y además poder utilizarlo pasándole desde cualquier lugar, la base de conocimiento y el conjunto de reglas que se quiera consultar a través de un archivo y de esta forma dar respuesta a nuestras peticiones.
- Debe ser seguro, no permitiendo que se violen la semántica dinámica de sus clases.

2.1.3 Determinación de los requerimientos Prolog

Para el buen funcionamiento del agente los datos de entrada deben estar de la siguiente manera:

- La base de conocimiento debe estar compuesta, por afirmaciones, hechos y reglas siguiendo las especificaciones sintácticas de prolog. En el caso de las reglas se recomienda comentar su funcionamiento.

- El Prolog recibe reglas o afirmaciones, el nombre o los argumentos que estos poseen no deben involucrar, paréntesis o comas.
- Los átomos utilizan únicamente letras minúsculas y las variables comienzan con letras mayúsculas.
- El motor de inferencia no reconoce cadena de String o argumentos dentro de comillas o tildes.
- Una regla puede tener como máximo 10 variables.
- La regla debe estar escrita en una línea continua, el programa no reconoce reglas o afirmaciones que están en varias líneas.
- No se puede utilizar el operador ! (negación), ya que el swi-prolog no lo acepta, en su defecto se debe aplicar la leyes de Morgan como caso equivalente.
- No hay que dejar caracteres en blanco antes del nombre de una regla o hecho a consultar.

2.1.4 Determinación de los requerimientos de hardware y software para la implementación del agente

- El agente hasta ahora puede correr en la plataforma linux, ya que uno de los programas requeridos funciona hasta este momento en esta plataforma.
- El agente es programado en Java y la base de conocimiento con la cual puede funcionar deben estar programadas en prolog.
- Para manipular Prolog dentro de java se utilizan clases del paquete JPL, permitiendo implementar de manera muy sencilla consultas dentro de java a la base de conocimiento que posea el agente en ese momento para sus respectivas consultadas.
- El agente puede ser utilizado por aquellos programas que estén realizado en Java, permitiendo de manera directa invocar los métodos del agente que permitan cargar la base de conocimiento en la base de datos de prolog y realizar las respectivas consultas.
- Se necesita SWI-PROLOG, versión 3.0.0 o mayor ya que el JPL es desarrollado para esta versión de Prolog.

2.2 Detalles de diseño

El agente diseñado para la plataforma de simulación GALATEA, permite involucrar las interacciones intencionales que se desarrollan con los agentes que a través de la orientación por objetos no incorpora un modelo interno que le permita al agente comunicarse con el ambiente, ofreciendo a GALATEA mecanismos de representación del mundo real a modelar permitiéndole cumplir su objetivo enriqueciendo los lenguajes y las herramientas de simulación, con las abstracciones necesarias para modelar agentes. Por otro lado, el motor de inferencia construido también puede ser utilizado en cualquier programa utilizando sus métodos, o de manera independiente a través de la base de conocimiento y el conjunto de consultas que se le va a realizar.

El agente pretende alcanzar las siguientes metas:

- Incorporar los avances en la teoría de modelado y simulación de sistemas multiagentes a través del modulo agente, a la plataforma de Simulación GALATEA.
- Construir un agente de manera genérica y de fácil uso.
- Implementar a través de Java consultas a prolog de manera muy sencilla.

Gracias a la utilización del lenguaje Java como lenguaje para programar el agente es posible que el agente sea independiente de la plataforma física en donde se ejecute y gracias a la orientación a objeto su funcionamiento es independiente del resto de los modulos del simulador GALATEA. De esta forma puede ser ejecutado en paralelo con el simulador.

2.2.1 Galatea

Los componentes de GALATEA funcionan también de manera independiente y la iteración de estos a través del tiempo se puede ver como la participación de una serie de objetos tales como: actividades recursivas de activación, recorrido de la red de nodos y la activación de los agentes que participan en el sistema. Los componentes asociados al proceso de simulación se pueden ver en la figura 1.1 y la participación del motor de inferencia en este se puede ver como:

FEL. Contiene el conjunto de eventos que tienen lugar en la simulación del sistema, estos están ordenados cronológicamente. Muchos de estos eventos son las peticiones realizadas al motor de inferencia que son enviadas por medio de la **interface** para que este la anexe a la **FEL**.

Control. Representa el control de proceso de simulación y solicita la programación de eventos que incluya la activación del agente.

Network. Esta representado por un conjunto de subsistemas (nodos) presentes a simular y controla la activación del agente.

Interface. Permite enlazar el agente y el simulador, a través del **interface** se realizan las diferentes consultas al agente que luego son anexadas a la **FEL**.

Agents. Este tiene la función de controlar las actividades propias del agente, tales como: razonar, observar y actuar.

razonar. Cuando toma las decisiones a través del motor de inferencia, para asimilar percepciones, observaciones, metas, creencias y preferencias.

observar. Percibir el ambiente, del ambiente el motor de inferencia toma el conjunto de reglas o hechos que luego razona.

actuar. Intenta modificar el ambiente de acuerdo a las decisiones tomadas por el motor de inferencia, la salida del motor de inferencia es enviada a la **interface** para que sea anexada a la **FEL** y de esta manera trate de modificar el ambiente.

2.3 Detalles del Agente GALATEA

El agente está diseñado por un conjunto de componentes entre los que destaca la **Adquisición de conocimiento** y **aprendizaje**. Este componente incluye un motor de aprendizaje y un motor de inferencia cuyo comportamiento esta asociado al formalismo en el cual se presenta la base de conocimientos, así como también la metodología para construir la base de conocimiento automáticamente.

Este agente de tipo racional posee dos características importantes: la adquisición de conocimiento y la capacidad de aprendizaje a la que se le agrega alguna capacidad elemental para resolver problemas intrínsecos del aprendizaje y una base de conocimiento.

2.3.1 Componentes

Controlador

El componente controlador se encarga de ejecutar algunas operaciones independientes de dominio tales como comparar la sintaxis y las reglas, mientras que el manejador de la base de datos permite el acceso y la modificación del contenido de la base de datos.

Además, sirve de canal entre la interface y el motor de inferencia.

La representación del conocimiento se refiere a la correspondencia entre el dominio de aplicación externa y el sistema de razonamiento simbólico. En otras palabras,

es la información que tiene el agente del mundo externo. Esta representación esta conformada por una estructura de datos para almacenar la información y los métodos que permiten manipular dicha estructura.

La representación del conocimiento y el razonamiento del agente se fundamenta en la base de conocimientos y a continuación se presentan algunos detalles de su implementación.

Base de Conocimientos

Contiene la información disponible para el agente. Esta información esta almacenada por medio de hechos y reglas, que luego es utilizada para proporcionar las soluciones correctas a las diferentes consultas realizadas por el motor de inferencia.

Existen muchas alternativas de implementación de este almacén de conocimientos. Sin embargo, en el desarrollo de una base de conocimientos se pueden distinguir tres fases:

1. Incorporación
2. Refinamiento
3. Reformulación

En la etapa de incorporar conocimiento se define el conocimiento básico del agente.

En la fase de refinamiento se extiende y se depura la base de conocimientos.

Durante la fase de reformulación se organiza la base de conocimientos para mejorar la eficiencia del controlador a la hora de contrastar las reglas.

La descripción del agente incluye varios aspectos y hasta ahora hemos presentado su arquitectura interna, pero en [Dávila:1997] se estudia toda una teoría de como se puede describir un agente utilizando lógica.

Motor de inferencia

El motor de inferencia interactúan de manera directa con la base de conocimientos, utilizando las clases de JPL que permite manipular dentro Java, la base de conocimientos realizada en prolog. De esta forma al comenzar la simulación del sistema, es inicializado el JPL y automáticamente es cargada la base de conocimientos en memoria principal del computador, luego el motor de inferencia queda esperando cada uno de los datos enviados por el simulador a través de la **interface**, el motor de inferencia automáticamente recibe la orden, consulta la base de conocimientos y envía su respuesta a la **interface** por medio del controlador para que el simulador tome estos datos, que pasan a formar parte de la listas de influencias para la simulación

que se esta ejecutando, de esta forma y durante la simulación el motor de inferencia esta alerta de cualquier llamado que pueda realizar la **interface**.

Como el objetivo principal es construir un motor de inferencia para un agente que pueda ser incorporado a la plataforma de Simulación GALATEA, siguiendo todas las pautas descritas en [Uzcátegui:2002] donde se describe la arquitectura de la plataforma.

El motor de inferencia cumple con los requerimientos anteriores, y se muestra en el diagrama de casos de uso de la figura 2.1.

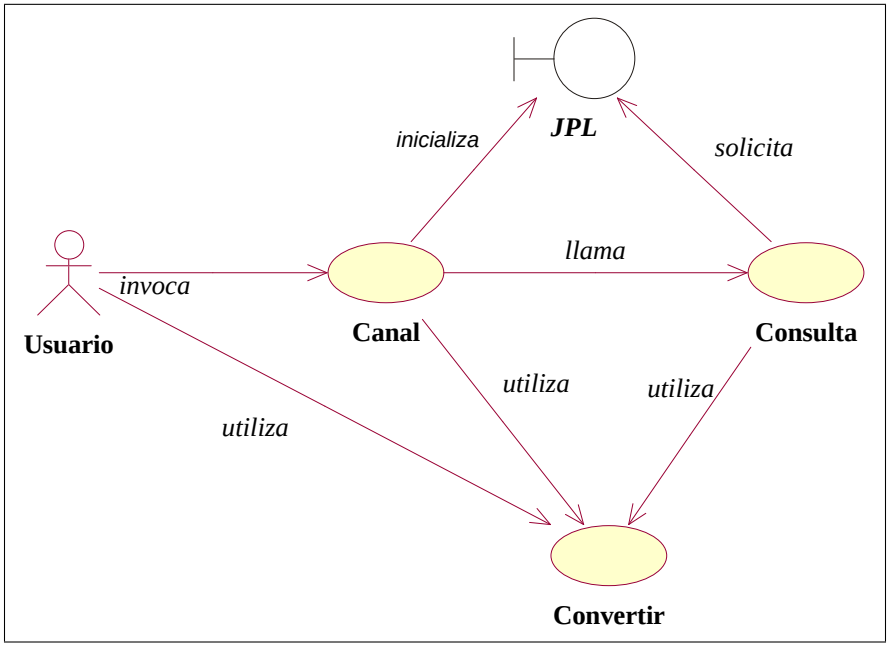

Figura 2.1: Diagrama de Caso de Uso

Este sistema puede ser descrito con dos actores y tres casos de uso.

Actores

1. **Usuario.**
   Actor que representa a los usuarios del sistema. Se tiene tres tipos de usuarios:

   (a) **GALATEA** dentro de GALATEA, la interface le pasa la base de conocimiento y la lista de reglas a consultar al agente.

   (b) **Usuario Externo** ejecuta desde consola, el nombre de la base de conocimiento, y el nombre del archivo donde están las diferentes consultas que se le va a pasar al Agente.

(c) **Usuario Activo** Le pasa la base de conocimiento e interactúan con el agente haciendo consultas por teclado.

2. **JPL.**
   Actor que representa la interfaz con *SWI-Prolog* en plataforma Java, permite cargar en Prolog la base de conocimientos y realizar las respectivas consultas a Prolog.

Casos de Uso

1. **Canal**

   Caso de uso que describe el proceso de dar formato a las diferentes reglas o hechos a consultar y retorna por cada regla valida los valores de las variables de la regla y para cada hecho retorna el estado (verdadero o falso).

   - Entradas

     Base de Conocimiento: Recibe el nombre de la **base de conocimiento**, con este inicializa el **JPL** y además le pasa el nombre de la **base de conocimiento** para que el **JPL** la cargue dentro de Prolog.

     Consultas: El Usuario, le envía el nombre del archivo donde está el conjunto de consultas que se le puede realizar al Agente, estas consultas pueden ser hechos o reglas de los cuales el Agente debe dar las respectivas respuesta.

   - Salidas

     Salidas generadas por pantalla: En este caso, imprime por pantalla la salida de cada una de las consultas realizadas, infiriendo a través de la base de conocimiento la primera respuesta que encuentra y despliega por pantalla en el caso de una regla los valores de las diferentes variables de la regla, o en el caso de un hecho el estado de este (verdadero o falso)

     Archivo General: Escribe en un archivo general todas las salidas retornadas por las diferentes consultas realizadas al Agente.

     Registro de la última consulta: En un archivo, aparece el registro de la última consulta realizada al Agente.

   - Escenario Principal
     (El Usuario le pasa al Agente el nombre de la Base de Conocimiento y el nombre del archivo donde están las consultas a realizar)

     Pre-condiciones

- La base de conocimiento debe estar programada en Prolog.
- El archivo que la contiene debe ser de extensión *pl*(ejemplo: concocimiento.pl)
- El conjunto de consultas a realizar, debe ser válida dentro de la base de conocimiento.
- El nombre del archivo donde están las consultas a realizar debe tener extensión *txt* (ejemplo: *consulta.txt*)

Pasos

(a) Se debe ejecutar dentro de la consola de trabajo, el programa principal que manipula el Agente, pasándole el nombre de la base de conocimiento sin la extensión y el nombre del archivo donde se encuentra las diferentes consultas sin la extensión. Ejemplo:

java -cp motor.jar Tuberia

Post-condiciones

- El programa despliega por pantalla, la salida de cada una de las consultas.
- Escribe la salida de todas las consultas dentro de un archivo.
- La última consulta también es registrada en un archivo.

- Escenario Alternativo:

  Utilización de los métodos personales, dentro de nuestros propios programas personales.

  Pre-condiciones

  - Al realizar las consultas al Agente, se debe validar el arreglo donde se retornará la salida generada por la consulta.
  - Importar dentro del programa donde se utilice sus métodos, el paquete motor.
  - Al realizar las consultas al Agente, se debe validar el arreglo donde se retornará la salida generada por la consulta.

  Pasos

  (a) Se debe inicializar el JPL y cargar la base de conocimiento dentro de la base de datos de prolog utilizando el método de inicialización de **Canal**.

(b) Invocar el método indicado que permita realizar la consulta al Agente, pasándole la consulta a realizar.

Post-condiciones

– La salida generada, en cada consulta es devuelta en un arreglo de tipo *Object*. El usuario deberá manipular el arreglo para dar formato según la salida deseada.

2. **Convertir**

Caso de uso que describe el proceso de manipular y acondicionar cada una de las consultas al **JPL** como también la salida que pudiera generar proporcionando un formato válido.

- Entradas

Canal: A través de **Canal** recibe de canal la consulta a realizar y convertir tiene la responsabilidad de generar un formato válido para almacenar la salida retornada por la consulta realizada por parte de **Canal**.

Consulta: Recibe de parte de **Consulta**, una serie de peticiones, que le permite acondicionar el formato correcto a las consultas realizadas al **JPL**.

- Salidas

Nombre de regla o hecho: Le puede enviar a **Consulta** el nombre de la regla o hecho a consultar.

Cantidad de argumentos: Le da información a **Consulta** sobre la cantidad de argumentos que pudiera tener un hecho o una regla

Cantidad de variables de una regla: Le envía a **Canal** o a **Convertir**, la cantidad de variables que pudiera tener una regla.

Relación de Argumento: Le envía a Consulta, todos los argumentos de una regla o hecho previamente identificados como variables, números o átomos.

- Escenario Principal

(Colabora con Canal)

Pre-condiciones

– La consulta pasadas a **Convertir** para que este las valide debe tener las especificaciones sintácticas de Prolog. Tales es el caso de el uso de variables o átomos (las variables comienzan con letra mayúscula y los átomos comienzan con letra minúsculas).

- El nombre de las reglas o hechos a consultar o los argumentos que estos poseen no deben involucrar, paréntesis o comas.

Pasos

(a) Invocar a **Convertir**, para validar el arreglo donde se va almacenar la salida generada en la consulta.

- Escenario Alternativo

(Colabora con Consulta)

Pre-condiciones

- Las consultas que se les pasa a **Convertir** para que las valide o las acondicione para que sean pasadas al **JPL**, deben cumplir las mismas condiciones del escenario anterior.

Pasos

(a) Le envía a **Consulta** el nombre de la regla o hecho a consultar.
(b) Le envía a **Consulta** la cantidad de argumentos que pose la consulta.
(c) Clasifica los diferentes argumentos que pose la consulta que se le va a realizar al **JPL**.

Post-condiciones

- Validar las variables donde se van almacenar las diferentes peticiones realizas por **Consulta** a **Convertir**.

- Escenario Alternativo

(Puede ser utilizado en programas personales)

Pre-condiciones

- Debe importar el paquete motor, dentro del programa personal.
- Las consultas que se le realice a convertir deben cumplir con las especificaciones sintácticas de Prolog.

Pasos

(a) Llamar a Convertir que permite validar el arreglo donde se va a almacenar la salida generada de la consulta realizada al Agente.

Post-condiciones

- Los valores retornados por Convertir se deben almacenar en variables que sean validas al formato enviado.

3. **Consulta**

Caso de uso que describe el proceso de llevar a cabo las diferentes consultas a Prolog a través de **JPL**, recibe las diferentes reglas o hechos a consultar y

retorna por cada regla valida los valores de las variables de la regla y para cada hecho el estado (verdadero o falso).

La consulta que recibe por parte de **Canal**, son manipuladas y acondicionas con la ayuda de **Convertir**, para realizarle a *Prolog* a través del **JPL**, las diferentes consultas a la base de conocimiento que tiene el agente en ese momento para emitir sus respuestas.

- Entradas

  Consulta a realizar: Recibe de parte de **Canal**, la consulta a realizar.

  Validación: Recibe de parte de **Convertir** las diferentes consultas acondicionadas 'de tal manera que **Consulta** le pueda dar el formato apropiado para que el **JPL** las pueda interpretar.

  Respuesta del JPL: El **JPL** le devuelve a **Consulta** la respuesta de la consulta realizada.

- Salidas

  Acondicionamiento de la consulta: **Consulta**, le envía la consulta a **Convertir**, para que este le de un formato valido, que luego es utilizado para acondicionar la consulta realizada al **JPL**.

  Envío de la consulta al JPL: **Consulta**, le envía al **JPL**, en un formato válido la consulta a realizar a la base de conocimiento por medio de Prolog.

  Respuesta de la consulta: **Consulta**, le envía respuesta de la consulta realizada a **Canal**, para que este le notifique al usuario el resultado arrojado por la consulta.

- Escenario principal

  (interfaz)

  Pre-condiciones

  – Recibir, una regla o consulta por parte de **Canal** y que esta sea valida dentro de la base de conocimiento a consultar.

  Pasos

  (a) Toma la consulta enviada por parte de **Canal**.

  (b) Busca el nombre de la regla o hecho (nr) que se va a consultar, por medio de **Convertir**.

  (c) Busca la cantidad de argumentos (na) que tiene la regla, realizándole la petición a **Convertir**.

  (d) Le pide a **Convertir**, la identificación de los diferentes argumentos (variable, número o átomo) de la regla o hecho a consultar (r).

(e) Válida cada unos de los argumentos del hecho o regla a consultar.

(f) Realiza en un formato válido la consulta al **JPL**.

Ejemplo

En este caso, la consulta que recibe por parte de Canal, son manipuladas y acondicionas con la ayuda de Convertir, para realizarle a Prolog a través del JPL, las diferentes consultas a la base de conocimiento que tiene el agente en ese momento para emitir sus repuestas.

2.3.2 Comportamiento del modelo Agente

Existen dos formas que permiten interactuar con el agente y a continuación presentamos la iteración a lo largo del tiempo entre cada uno de sus componentes:

1. **Pasándole el nombre de la Base de Conocimiento y el nombre del archivo donde están las consultas a realizar.**

   La iteración del Usuario con el motor de inferencia y cada uno de sus componentes internos se muestra en el diagrama de la figura 2.2.

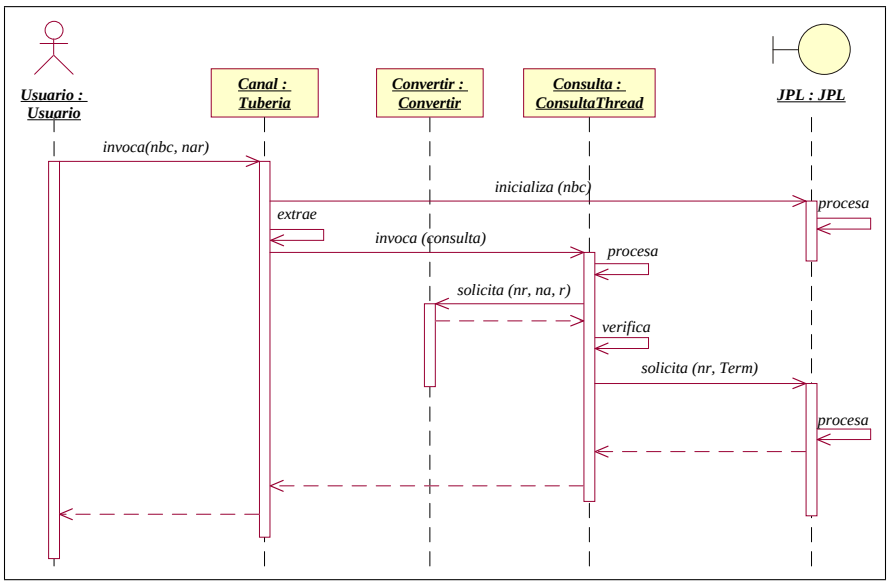

Figura 2.2: Diagrama de Secuencia

Objetos

(a) **Usuario**

Usuario que interactúa con el Agente, le envía el nombre de la base de conocimiento y el nombre del archivo donde están las diferentes consultas.

(b) **Canal**

Clase Tuberia

Tiene la responsabilidad de inicializar el **JPL**, cargar la base de conocimiento en prolog y canalizar todas las consultas enviadas por el **Usuario** y dar respuesta de estas.

(c) **Consulta**

Clase ConsultaThread

Le envía al **JPL** la consulta a realizar en el formato adecuado, para que el **JPL** pueda consultar la base de datos de Prolog.

(d) **Convertir**

Clase Convertir

Da respuesta a las diferentes solicitudes realizadas por **Canal** y **Consulta**, su principal objetivo es permitirle a **Canal** validar la salida generada de la consulta y a **Consulta** dar formato de las diferentes consultas que se realizan al **JPL**.

(e) **JPL**

Conjunto de clases de *Java* que permite realizar consultas muy sencillas dentro de *Java* a *Prolog*

Los pasos que siguen durante la validación, asociando un objeto a cada actor o caso de uso es el siguiente:

(a) **Usuario** *invoca* a **Canal** pasándole el nombre de la base de conocimiento y el nombre del archivo donde están las consultas realizadas al Agente.

(b) **Canal** *inicializa* al **JPL** y carga la base de conocimiento dentro de la base datos Prolog.

(c) **Canal** *extrae* del conjunto de consultas dentro del archivo, la primera consulta que encuentra. Discrimina entre un hecho y una variable e inicializa el arreglo donde retornará el estado del hecho o los valores de las variables de la regla que se le consultaran al Agente.

(d) **Canal** *invoca* a **Consulta**, y este procesa la consulta ante el **JPL**.

(e) *procesa* la consulta realizada, comenzando a acondicionar la consulta a realizar en un formato que lo pueda entender el JPL.

(f) **Consulta** *solicita* a **Convertir** el nombre de la regla que se quiere consultar (nr), la cantidad de argumentos que posee (na) y la clasificación de cada uno de los argumentos por su tipo (r).

(g) **Convertir** retorna la solicitud hecha a **Consulta**.

(h) **Consulta** *verifica* la cantidad de variables que posee la regla o los argumentos que pudiera tener y se los asigna a un arreglo de tipo *Term*.

(i) **Consulta** *solicita* al **JPL** la consulta a *Prolog*, pasándole el nombre de la regla (nr) o hecho a consultar y el arreglo($Term$) donde están almacenados los átomos, variables o números de la consulta.

(j) **JPL** procesa la consulta ante *Prolog*

(k) **JPL** devuelve la solución a **Consulta** si la solicitud realizada por **JPL** al *Prolog* fue satisfactoria.

(l) **Consulta** retorna la respuesta a **Canal**.

(m) **Canal** escribe la respuesta en un archivo donde están todas las salidas generadas por las consultas realizada, también en un archivo donde se almacena la salida de la ultima consulta y también imprime por pantalla la salida generada.

2. **Desde nuestros propios programas personales, ejemplo GALATEA**

   En este segundo escenario las principales diferencia que se tienen con el escenario anterior es que el **Usuario** que interactúa con el Agente, en nuestro caso es **Galatea** envía por medio de la interface del Simulador, le envía al controlador del agente el nombre de la base de conocimiento a consultar y una serie de consultas y el Agente le envía las diferentes respuesta, para que el Simulador las tome en cuenta durante la simulación de algún sistema.

   En este caso el **Usuario** (GALATEA), *inicializa* el **JPL** como también la base de conocimiento en prolog, y además, realiza cada una de las consultas a la base de conocimiento de manera directa, solicitando información a **Convertir** para saber si la consulta a realizar es un hecho o una regla, inicializar el arreglo donde retornará la salida de cada una de las consultas realizadas al **JPL**. En el diagrama 2.3 se destacan las siguientes colaboraciones.

2.4 Implementación del prototipo

La implementación de este prototipo cumple con los lineamentos descritos en el diseño del sistema.

Justificación

Esta implementación se realizó construyendo un **paquete motor** que le permite adaptarse de manera simple al simulador GALATEA, y gracias a la programación orientado a objeto, los métodos de sus clases pueden ser utilizados de manera independiente dentro de GALATEA, y basado en la especificación proporcionada por la

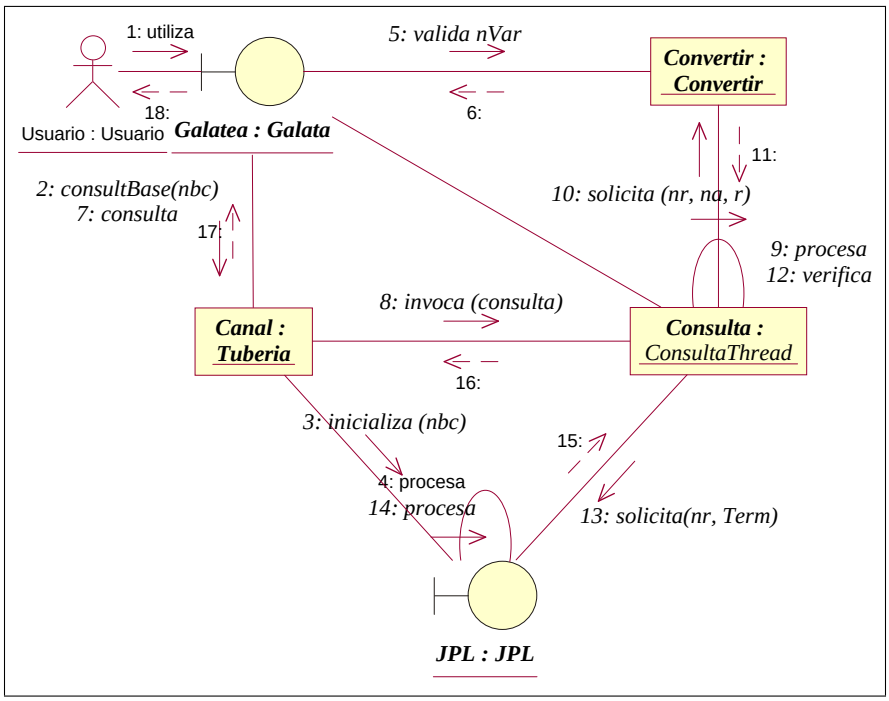

Figura 2.3: Diagrama de colaboración para el Agente

teoría de simulación multi-agentes [Dávila–Tucci:2000], se le puede asociar la ejecución de un motor de inferencia a cada agente y de esta forma, estos motores pueden ejecutarse concurrentemente con el simulador principal, y así mientras el simulador controla la evolución física del sistema, los motores de inferencia le proporciona el ciclo de percibir-razonar-actuar a cada agente.

La arquitectura del prototipo se puede representar en el siguiente diagrama de clases.

En la figura 2.5, se puede apreciar cada una de las clases que pose el JPL.

A continuación presentamos una breve descripción de cada una de las clases que integran el motor de inferencia

2.4.1 ConsultaThread

Paquete motor

El objeto **ConsultaThread** permite realizar la consulta de las diferentes reglas que se le quiera realizar a la base de conocimiento previamente cargada en la base de datos de *Prolog*, su salida se realiza por medio de un *PrintStream* (que escribe la salida en una tubería). El objeto thread permite que cada final de la tubería se

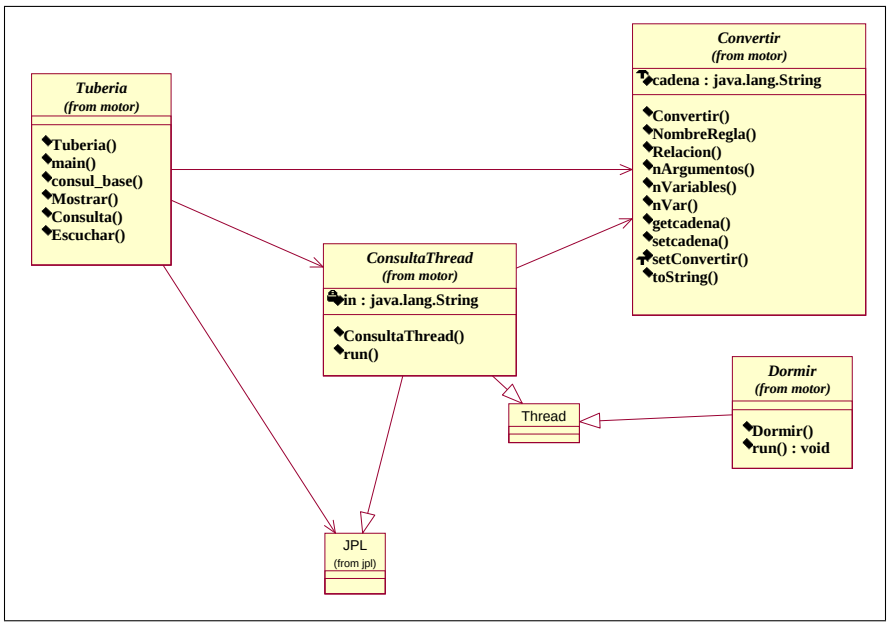

Figura 2.4: Diagrama de Clases

ejecute independientemente y evita que el método *main()* se bloquee si una de las tuberías se bloquea mientras espera a que se complete una llamada a *I/O*.

Métodos

run

En este método se declara un conjunto de variables entre las que están un arreglo de tipo *Variable*, clase perteneciente al paquete **JPL** que se utiliza para consultar reglas cuyos argumentos tengan variables, el arreglo de variables posee un conjunto de 10 variables que para nuestro caso es el número máximo de variables que debe poseer la regla a consultar, un arreglo de *term*, que es de tipo *JPL.Term*, otra clase del paquete JPL y que es utilizado para almacenar todos los términos de la regla a consultar, que después es utilizado para inicializar el objeto *query*, perteneciente al paquete **JPL** y permite realizar las diferentes consultas a Prolog.

Dentro del método **run** se crea un objeto de tipo **Convertir** llamado *conversor*, que es utilizado para manipular la cadena recibida por medio del Pipe para tener el nombre y los argumentos de la regla a consultar en la base de conocimiento. Primero se invoca el método *nr* del método convertir y se obtiene el nombre de la regla, luego se llama al método *na* de la clase **Convertir** para tener el número de argumentos de la regla y con este poder inicializar el arreglo term que tendrá todos los argumentos de la regla, inmediatamente se invoca el método *relacion (r)* que devuelve un arreglo de tipo string de dos dimensiones con todos los argumentos de la regla, luego de

inicializar el arreglo de *term* se busca la cantidad de variables que pudiera tener la regla invocando el método *nVar* de la clase convertir. Al tener el número de variables que tiene la regla, se comienza a asignar cada uno de los argumentos de la regla al arreglo de *term*, y a medida que se va asignando cada uno de los argumentos se verifica si es una *variable*, un *átomo* o un *número*, y de esta forma poder inicializar el objeto correspondiente según sea el caso (átomo, variable o entero) a la hora de asignárselo al arreglo *term*.

Luego de tener todos los argumentos asignados se crea un objeto de tipo *query* perteneciente al paquete JPL, el cual es inicializado con el nombre de la regla y la lista de argumentos que están dentro de term para realizar la consulta, al tener a *query* inicializado se realiza la consulta en **JPL** y luego se verifica si la consulta realizada tubo éxito o no, si no tubo éxito el programa imprime por pantalla que la consulta fallo usando el objeto *query*, si tubo éxito y la consulta es verdadera, se verifica si la consulta realizada posee algún tipo de variable, si posee algún tipo de variable, se toma la primera solución que se logra inferir en la base de conocimiento de Prolog y se muestra el valor de cada una de las variables consultadas en la base de conocimiento, si tubo éxito y no posee variables el programa arroja *true* de lo contrario arroja *false*. Si la regla consultada no existe en la base de conocimiento el programa genera un error ya que al realizarse la consultar en **JPL**, *swi-prolog* genera un mensaje de error informando que no encuentra el predicado.

2.4.2 Dormir

Paquete motor

El objeto **Dormir** permite tener corriendo el motor de inferencia cuando se quiere consultar por medio de reglas en personales.

Métodos

run
Utiliza un lazo de repetición infinito

2.4.3 Tuberia

Paquete motor

Es la clase principal que permite dar funcionamiento al motor de inferencia, en esta clase se manipulan todos los métodos necesarios para realizar las diferentes consultas a *Prolog* por medio del **Jpl**, y esto se puede realizar a través del método principal *main* o utilizando cada uno de los métodos adjuntos que posee la clase, en los diferentes programas que nosotros quisiéramos crear.

Métodos

main

Este método recibe un arreglo de tipo *string*, que incluye el nombre de la base de conocimiento que se quiere consultar y el nombre del archivo donde están las diferentes reglas a consultar en la base de conocimiento de Prolog.

Este método abre un archivo de entrada, de tipo *FileReader* inicializado con las reglas que se quiere consultar, luego se llama al método *consulab*, que es inicializado con el nombre de la base de conocimiento, luego se utiliza el método *readLine* para leer cada una de las reglas del fichero de entrada, y al leerse la regla se crea un objeto de tipo **Convertir** y se utiliza el método *nVar* del objeto **Convertir** para saber la cantidad de variables que posee la regla y poder inicializar el vector *term*, si el número de variables resultara cero, a la variable *nvar* se le asigna 1, ya que en este caso se esta consultando un hecho, y gracias al método *mostrar* se le asignará verdadero (*true*) o falso (*falso*) en el caso de un hecho, o los valores de las variables en el caso de ser una regla, si la regla llegara a ser falsa, el arreglo retornado por el método *mostrar* devolvería en su primer término falso y en el resto nulo, entonces al manipular el arreglo arrojado por el método mostrar se debe validar a la hora de manipular cada uno de los términos si es nulo o no, y esto lo hacemos en este método al imprimir por pantalla y escribir en el archivo *salidap.data* cada uno de los elementos del arreglo *term*.

consultab

El método **consulb**, recibe el nombre de la base de conocimiento y permite inicializar al **JPL**; además permite cargar el archivo de la base de conocimiento en la base de datos de *Prolog*.

mostrar

El método **mostrar** recibe una regla, que es la utilizada para consultar la base de conocimiento, el resultado devuelto lo escribe en un archivo y también lo devuelve en un arreglo de tipo *Object*, este método comienza creando el archivo de salida donde almacenara la repuesta de la regla a consultar. Crea un objeto de tipo **Convertir** y busca la cantidad de variables que posee la regla, luego al tener la cantidad de variables se inicializa el arreglo de términos donde se almacenará los diferentes valores de las variables de la regla a consultar, luego invoca al método **consulta** que es inicializado con la regla que se quiere consultar, **consulta** devuelve un archivo de tipo *Reader* y luego se crea un archivo de tipo *Reader* para leer cada una de las líneas del archivo y asignar los valores obtenidos en el arreglo de *term* donde se va a tener almacenado

los valores de la variable y al realizar este proceso se va escribiendo en el archivo de salida.

consulta

El método **consulta**, toma un *String* que contiene la regla que se le va a consultar a prolog. Consulta crea dos objetos *PipedWriter* (*pipeOutput*) y (*pipeIn*), conecta el archivo de salida con la entrada e inmediatamente inicializa a **ConstultaThread** con el canal de salida de tipo *pipeOutput* y la regla a consultar, el resultado arrojado, que es el resultado de la consulta dentro de *Prolog* es devuelto por medio de la tubería al método mostrar quien fue el que lo invocó.

Devolviendo un fichero de tipo *reader*, con los valores arrojados en la consulta.

escuchar

Llama al Thread **Dormir** que permite tener corriendo al motor de inferencia, en programas personales.

2.4.4 Convertir

Paquete motor

métodos

nombreRegla (nr)

Este método recibe un *String* y devuelve un *String*. Utiliza el método *substring* de la clase *String* para cortar la cadena recibida y toma únicamente el nombre de la regla cortando la cadena hasta la posición donde encuentra el paréntesis que agrupa los argumentos de la regla. Luego devuelve la subcadena cortada que es el nombre de la regla.

relacion (r)

Introduce cada uno de los argumentos de la regla en un arreglo, el método recibe un *String* y devuelve un *String* de dos dimensiones cuya longitud es igual al tamaño de argumentos que posee la regla. En la primera columna del string esta los diferentes argumentos de la regla, en la segunda columna del *string* se encuentra la identificación de cada uno de los argumentos que posee la regla, identificándolo con 0 los *átomos*, 1 todos los argumento que son *variables* y con 2 aquellos argumentos que son *números*.

Funcionamiento interno

La cadena que recibe el método **relación** es utilizada para tomar una subcadena que se encuentra delimitada por los paréntesis que posee la regla. La cadena resultante posee todos los argumentos de la regla separadas por comas, entonces; se llama el método *na* para saber la cantidad de argumentos que posee la regla e inicializar el arreglo de dos dimensiones, luego cada uno de los argumentos se van asignando a la primera columna del arreglo de dos dimensiones, luego se toma el primer carácter del argumento asignado y se verifica si es un número, una variable o un átomo, si es un número en la segunda columna del arreglo se le asigna 2, si en una variable se le asigna 1 y si es un átomo se le asigna 0, luego al terminar la revisión de todos los argumentos de la regla, el arreglo de dos dimensiones es retornado con los argumentos de la regla previamente identificados.

nArgumentos (na)

Este método devuelve la cantidad de argumentos que posee la regla, el método recibe un cadena de tipo *strign*, que es en si la regla y devuelve un *entero*.

La cadena que recibe el método es picada y se toma lo que esta dentro de los paréntesis de la regla, luego la cadena resultante tiene todos los argumentos de la regla separados por comas, se cuenta la cantidad de comas que posee la cadena y se devuelve la cantidad de comas más uno, que es el número de argumentos que posee la regla.

nVariables

Devuelve todas las variables que posee la regla, el método recibe un arreglo de dos dimensiones y la cantidad de argumentos, **nVariables** devuelve un entero con la cantidad de variables que tiene la regla.

El método inicializa un arreglo de dos dimensiones con la cantidad de argumentos que tiene la regla luego asigna a este el arreglo que es pasado por referencia y comienza a revisar la segunda columna del arreglo y si posee un 1 el vector incrementa el contador *var* que contabiliza la cantidad de variables que posee la regla, y al recorrer todo el arreglo retorna el número de variables contabilizadas en la regla.

nVar

El método recibe la regla a consultar por medio de un *String* y devuelve un *entero*, con el número de variables que posee la regla.

De la cadena tomada se extrae una subcadena con todos los argumentos que posee la regla, luego va tomando cada uno de los argumentos de la regla y verifica si es una variable, si es una variable va incrementando *nvar* que contabiliza el número de variables que posee la regla que finalmente es devuelto al finalizar la revisión.

setConvertir

Inicializa la clase

2.5 Ejemplos

A continuación mencionaremos un conjunto de ejemplos que nos permite interactuar con el agente en diferentes escenarios, utilizaremos ejemplos muy sencillos que nos permita exponer de manera clara el funcionamiento del agente.

2.5.1 Familia

En el primer ejemplo, se muestra por pantalla el parentesco de cada uno de los miembros de una familia.

Se tiene una pequeña base de conocimiento, que representa un pequeño grupo familiar.

La misión del agente es mostrar el estado de los diferentes hechos que se le están consultando (conocer si Miguel es el padre de Manuel por ejemplo) y además poder inferir gracias a los hechos que contiene, el valor de las variables de las reglas válidas dentro de su base de conocimiento si se le llegara a consultar.

Para realizar esta consulta, se ejecuta la clase principal del motor de inferencia, pasando en este caso el nombre de la base de conocimiento, que tiene representado a través hechos y predicados el conjunto que compone todo el grupo familiar. También le es pasado el nombre del archivo que contienen las diferentes consultas que se van a realizar al motor de inferencia.

La base de conocimiento, que esta dentro del archivo *abuelos.pl* contiene la siguiente información:

$$padre(miguel, manuel).$$
$$padre(manuel, eliseo).$$
$$madre(miguel, lupe).$$
$$madre(lupe, amalia).$$

$$abuelo(X, Z) : -padre(X, Y), padre(Y, Z).$$
$$abuela(X, Z) : -madre(X, Y), padre(Y, Z).$$

Realizaremos la siguientes consultas que están el archivo *regla.txt*

$padre(miguel, manuel)$
$padre(manuel, eliseo)$
$madre(miguel, lupe)$
$madre(lupe, amalia)$
$Estasonlasreglas$
$abuelo(X, Z)$
$abuela(X, Z)$

dentro de la consola se teclea:

java -cp motor.jar Tuberia abuelos regla

En este caso el Agente, devolverá por pantalla la salida de cada una de las reglas a consultar de igual forma, escribirá en un archivo todas las salidas generadas y en otro archivo la salida de la ultima consulta.

La salida por pantalla y el contenido del archivo general es el siguiente:

$true$
$true$
$true$
$true$
$miguel$
$eliseo$
$false$

La información arrojada de las consultas realizadas no dice que los tres primeros hechos fueron verdaderos, de la primera regla *abuelo(X,Y)* consultada nos despliega *miguel* y *eliseo*, informando que *miguel* es abuelo de *eliseo*, pero la segunda regla *abuela(X,Y)* es *falsa*

También muestra en el archivo *salida.data* la respuesta de la última consulta que se realizó en este caso sería *false*, que es referida a la consulta *abuela(X,Y)*.

El objetivo de este primer ejemplo, es poder mostrar la utilización del motor de inferencia de manera genérica. Al ejecutar su clase principal se le pasa el nombre de la base de conocimiento y el nombre del archivo que contiene el conjunto de consultas que se van a realizar. También hay que notar que se puede ejecutar dentro o fuera del paquete motor en tal sentido hay que agregar *-cp motor.jar* cuando lo estamos ejecutando fuera del paquete motor.

Cuando el motor de inferencia se utiliza de esta manera, se tiene respuesta de

las diferentes consultas en un archivo, también le es mostrado por pantalla todas las salidas arrojadas y en otro archivo se tiene respuesta de la ultima consulta.

2.5.2 Espacio en Disco

Se desea conocer el espacio en disco de un computador (bastet servidor del laboratorio SUMA donde está almacenada la información de los usuarios) en temporada de vacaciones. Se instala un sistema experto que permita emitir alarmas para saber la cantidad de espacio disponible que pudiera tener el computador.

Las alarmas generadas por el sistema experto son las siguientes:

- El sistema experto emite un color verde, cuando la cantidad de disco ocupado esta por debajo del sesenta porciento, diciéndole al usuario que funciona de manera normal.

- El sistema experto, emite un color amarillo cuando la ocupación del disco duro se encuentra entre el sesenta y ochenta y cinco porciento, informándole al administrador del equipo que debe tener precaución al almacenar información en el computador.

- El sistema experto, emite una alarma de color rojo, informándole al administrador que la cantidad ocupada del disco esta entre el ochenta y seis y noventa y cinco porciento de su capacidad y en tal sentido, debe liberar información del disco.

- El sistema experto, dispara la alarma de color negro cuando el espacio de disco ocupado superó el noventa y cinco porciento de su capacidad, el tal sentido el administrador debe liberar el mayor espacio posible de disco para que el computador funcione normalmente de lo contrario el equipo deja de funcionar.

La misión del Agente, es disparar las diferentes Alarmas que se pudieran generar a causa del espacio en disco que posee el Servidor

Para simular el sistema descrito, se construyo un programa que genere aleatoriamente el espacio en disco que pose el computador, se construyo una base de conocimiento prolog que describiera las diferentes alarmas que genera el sistema experto El programa esta construido en Java, utiliza los métodos del motor de inferencia para realizarles las diferentes consultas a Prolog, sobre las alarmas que debe generar en determinado momento cuando el disco del computador tenga cierta capacidad.

En el método principal del programa, utiliza las clases **Tuberia** y Convertir del paquete **Motor**, la clase **Tuberia** utiliza el método *consul_base()*, que permite pasarle a la base de datos de swi-prolog la base de conocimiento e inicializa el **JPL** y también se utiliza el método *mostrar()*, método utilizado para realizar la consulta, de la clase

Convertir utilizamos el método *nVar()*, nos muestra la cantidad de variables que contiene la regla.

La base de conocimiento pose la siguiente información:

$$alarma(verde, G) : -G < 60.$$
$$alarma(amarillo, G) : -G >= 60, G =< 85.$$
$$alarma(rojo, G) : -G > 85, G =< 95.$$
$$alarma(negro, G) : -G > 95, G =< 100.$$

La salida arrojada por el agente en pantalla y escrita en el archivo salidap.data sería:

$$amarillo$$
$$amarillo$$
$$amarillo$$
$$verde$$
$$amarillo$$
$$verde$$
$$verde$$
$$verde$$
$$verde$$
$$negro$$

El agente deja de informar cuando el espacio en disco sobrepasa el noventa y ocho porciento de su capacidad.

El propósito de este ejemplo es poder implementar los diferentes métodos del paquete motor dentro de nuestro programas personales, para realizar consultas a Prolog. Hay que notar también, que la salida arrojada en cada una de las consultas, esta en un formato muy genérico (de tipo Object) y de fácil manipulación, al igual que en el ejemplo anterior la ultima consulta es almacenada dentro de un archivo.

Lo esperado de este ejemplo, es que el lector pueda ver cual es la forma y el orden de utilización de los métodos dentro de los programas personales.

2.5.3 Cupido

Cupido es el más famoso de los símbolos de San Valentín; todos conocen al niño que anda flechando corazones. Es ilustrado como un niño alado y armado con arco y flechas. Las flechas significan deseos, emociones y amor, y Cupido dispara esas flechas a dioses y humanos, provocando que se enamoren profundamente.

Cupido siempre ha tenido un papel importante en las celebraciones del amor. En

la antigua Grecia, era conocido como Eros, el joven hijo de Afrodita, la diosa del amor y la belleza. Para los romanos, él era Cupido, y su madre era Venus.

Existe una historia muy interesante acerca de Cupido y su novia mortal Psique en la mitología romana. Venus estaba celosa de la belleza de Psique, y ordenó a Cupido castigarla. Pero en vez de ello, Cupido se enamoró profundamente de ella. La tomó como esposa, pero como mortal, ella tenía prohibido verlo.

Psique era feliz hasta que sus hermanas la convencieron de ver a Cupido. Tan pronto como ella lo vio, Cupido la castigó abandonándola. Su castillo y sus jardines desaparecieron también. Psique se encontró sola en un campo abierto sin señales de nadie más, ni de Cupido buscando a su amor, fue hasta el templo de Venus. Deseosa de destruirla, la diosa del amor le dio una serie de condiciones, una más difícil y peligrosa que la anterior.

Como última instrucción, le había dado una pequeña caja, y le había dicho que la llevara al mundo submarino. Tenía que llevar un poco de belleza a Proserpina, la esposa de Plutón, y la misma había sido puesta en la caja. Durante su viaje, le fueron dados consejos para burlar los peligros. Psique estaba advertida de que no debía abrirla, pero la tentación la venció y abrió la caja. Y en lugar de encontrar belleza, encontró un profundo sueño que parecía la muerte.

Cupido encontró a su esposa en el suelo. Retiró el sueño mortal de su cuerpo y lo puso de nuevo en la caja. Cupido la perdonó, al igual que Venus. Los dioses, conmovidos por el amor de Psique hacia Cupido, la convirtieron en una diosa.

Hoy en día, Cupido y sus flechas se han convertido en el más popular de los símbolos del amor, y el amor es frecuentemente simbolizado como dos corazones atravesados con una flecha: la flecha de Cupido.

En este ejemplo el agente haciendo el papel de Cupido, tendrá la responsabilidad de flechar corazones y tratar que estos se enamoren, para tal fin, la información necesaria de sus víctimas amorosas estará almacenada dentro de su base de conocimientos.

$$
\begin{aligned}
&quiere\_a(maria, enrique).quiere\_a(juan, jorge).\\
&quiere\_a(maria, susana).quiere\_a(maria, ana).\\
&quiere\_a(susana, pablo).quiere\_a(ana, jorge).varon(juan).\\
&varon(pablo).varon(jorge).varon(enrique).hembra(maria).\\
&hembra(susana).hembra(ana).teme_a(susana, pablo).\\
&teme\_a(jorge, enrique).teme\_a(maria, pablo).\\
&quiere\_pero\_teme\_a(X, Y) :- quiere\_a(X, Y), teme\_a(X, Y).\\
&querido\_por(X, Y) :- quiere\_a(Y, X).puede\_casarse\_con(X, Y) :-\\
&quiere\_a(X, Y), varon(X), hembra(Y).puede\_casarse\_con(X, Y) :-\\
&quiere\_a(X, Y), hembra(X), varon(Y).
\end{aligned}
$$

El agente debe informar cada una las de peticiones (consultas) realizadas por el

usuario, desplegando por pantalla el nombre del par de corazones que serán flechados.

Este ejemplo se realizó con el fin de que el usuario pueda interactuar de manera directa cone le agente, pasándole consultas por teclado; pero estas consultas deben ser validas a la base de conocimientos que pose el agente y además siempre cumpliendo con las especificaciones sintácticas del lenguaje Prolog.

2.5.4 Trenes

Tomando el ejemplo del sistema de ferrocarril de [Uzcátegui:2002] sección 2.3.2, donde se desea simular un día de trabajo en un tramo de un sistema de ferrocarril simple. El ferrocarril tiene múltiples trenes y cada tren tiene varios coches. Las consideraciones del proceso son:

- El número de vagones que se ensamblan en cada tren es decidido por el fiscal del terminal principal donde parte cada uno de los trenes, pero se sabe que se encuentra entre 11 y 30 vagones.
- El tramo consta de dos estaciones: una estación de partida y una estación de destino.
- Los trenes salen de la estación de partida cada 25 minutos.
- El recorrido del tramo a estudiar dura 45 minutos, luego de los cuales el tren llega de la estación de destino.
- Los trenes permanecen en la estación destino en espera de un nuevo recorrido.
- El recorrido completo del ferrocarril es circular. Por ende, los trenes se encuentran tiempo completo viajando entre estaciones.

Para modelar el sistema asociamos el rol del fiscal a un Agente, que deberá decidir cual es la cantidad de vagones que deberá tener cada tren debido a la cantidad de pasajeros que este transportará al iniciar su recorrido. También se asocia una unidad de tiempo de simulación a un minuto. Los trenes serán representados como mensajes y la estructura del tramo ferroviario es representado a través de una red de tres (3) nodos:

Partida: Representando la estación de partida de los trenes. En este nodo el fiscal (Agente) debe decidir que cantidad de vagones tendrá cada tren, y cada tren (mensaje) se registra al llegar al sistema.

Ruta: Representa la ruta a seguir por los trenes.

Destino: Representa la estación de destino de los trenes. En este nodo se registran los mensajes (trenes) que salen del sistema.

La base de conocimientos posee la siguiente información:

$$
\begin{aligned}
&vagones(11, X) : -X =< 5.\\
&vagones(12, X) : -X >= 6, X =< 10.\\
&vagones(13, X) : -X >= 11, X =< 15.\\
&vagones(14, X) : -X >= 16, X =< 20.\\
&vagones(15, X) : -X >= 21, X =< 25.\\
&vagones(15, X) : -X >= 26, X =< 30.\\
&vagones(17, X) : -X >= 31, X =< 35.\\
&vagones(18, X) : -X >= 36, X =< 40.\\
&vagones(19, X) : -X >= 41, X =< 45.\\
&vagones(20, X) : -X >= 46, X =< 50.\\
&vagones(21, X) : -X >= 51, X =< 55.\\
&vagones(22, X) : -X >= 56, X =< 60.\\
&vagones(23, X) : -X >= 61, X =< 65.\\
&vagones(24, X) : -X >= 66, X =< 70.\\
&vagones(25, X) : -X >= 71, X =< 75.\\
&vagones(26, X) : -X >= 76, X =< 80.\\
&vagones(27, X) : -X >= 81, X =< 85.\\
&vagones(28, X) : -X >= 86, X =< 90.\\
&vagones(29, X) : -X >= 91, X =< 95.\\
&vagones(30, X) : -X >= 96, X =< 100.
\end{aligned}
$$

El Agente (fiscal) debe decidir la cantidad de vagones de un tren. Cuando este le toca arrancar.

El propósito de este ejemplo, es poder mostrar como el motor de inferencia puede funcionar dentro del simulador Galatea, de manera independiente y de manera paralela con este.

Algunas consideraciones

- Funciona de manera genérica, puede recibir únicamente el nombre de la base de conocimiento y el archivo que se quiere consultar.

  Una gran ventaja.. "*Puede servir de herramienta para cursos donde queramos aprender lenguajes lógicos (Prolog), y no tenemos que programar otra cosa para su implantación*"

- El Agente puede realizar grandes funciones dentro de nuestros programas personales, solo queda de un poquito de creatividad para lograrlo, un ejemplo palpable "El sistema experto"

- El Agente puede interactuar de manera directa con el usuario, pasándole consultas por teclado que sean validas dentro de la base de conocimiento que él pose.

2.5.5 Cartero

Se desea simular un día de trabajo de un cartero, para el uso de su paraguas. Las consideraciones del proceso son: o El cartero (mensajes) presta sus servicios durante 250 unidades de tiempo que representa un día de trabajo del cartero.

- El cartero permanece en su lugar de trabajo entre 5 a 25 unidades de tiempo.
- Durante su recorrido el cartero decide si saca o no su paraguas, y su recorrido puede tardar de 15 hasta 35 unidades de tiempo, este tiempo de recorrido se modela según una función uniforme aleatoria
- El cartero sale de su lugar de trabajo, realiza su recorrido hasta llegar a su destino.

Para modelar el sistema descrito, utilizamos un agente. Para decidir el tiempo que tarda el cartero en salir de su sede principal, el tiempo de recorrido para entregar todas sus encomiendas. Otro agente que nos permite simular el clima para saber si el cartero saca o no su paraguas. Se utilizan distribuciones de variables aleatorias que según el valor arrojado, cada uno de estos agentes decide que acción debe ejecutar por medio de su motor de inferencia.

La trayectoria realizada por el cartero, se lo asociamos a una red de tres nodos:

Partida. Representa la salida del cartero de su sede principal.

Recorrido. Representa el recorrido que realiza el cartero, durante el mismo, el cartero decide si saca o no su paraguas.

Destino. Representa el final del recorrido del cartero.

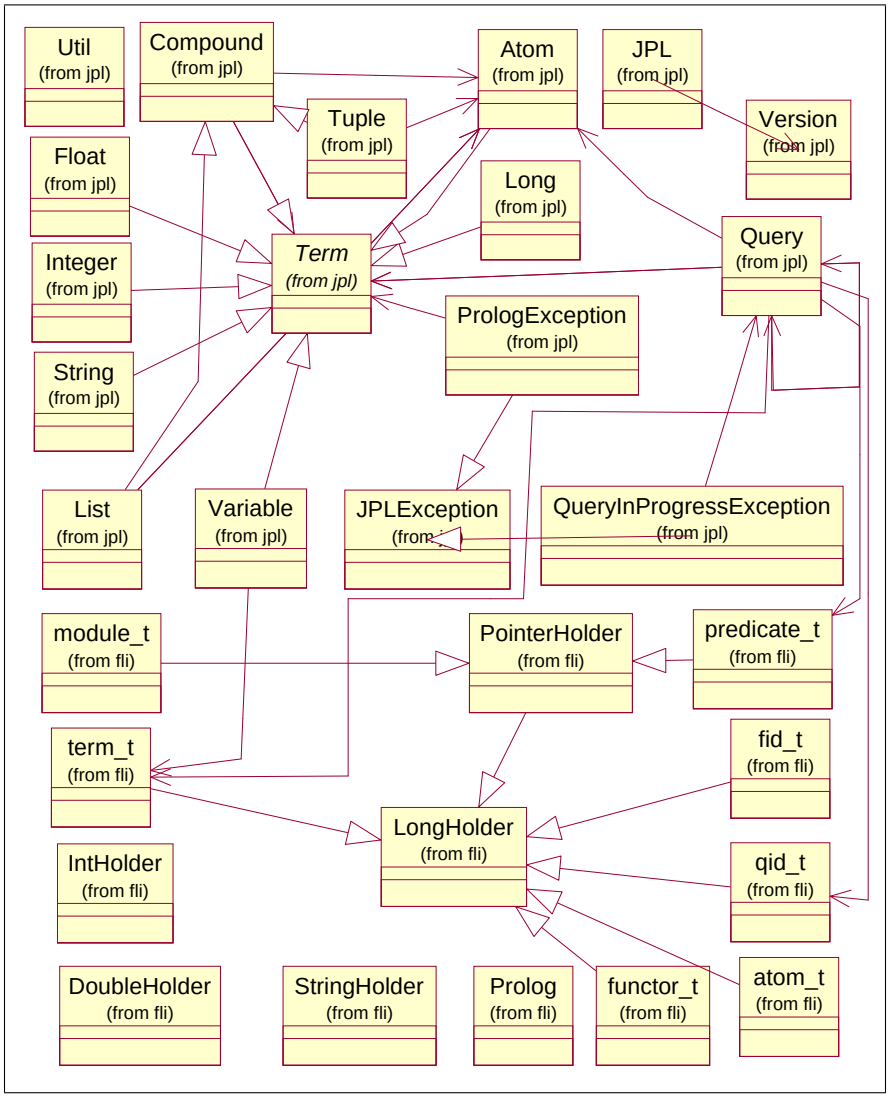

Figura 2.5: Arquitectura interna del JPL

Referencias

[Cabral:2001] Cabral, M. (2001). Prototipo del módulo GUI para la plataforma de simulación galatea. Tesis de Grado, Escuela de Ingeniería de Sistemas. Facultad de Ingeniería, Universidad de Los Andes. Mérida. Venezuela.

[Dávila:1997] Dávila, J. A. (1997). *Agents in Logic Programming.* Tesis PhD , Imperial College of Science, Technology and Medicine., London, UK.

[Dávila–Tucci:2000] Dávila, J. A. y Tucci, K. A. (2000). Towards a logic-based, multi-agent simulation theory. En *International Conference on Modelling, Simulation and Neural Networks [MSNN-2000]*, páginas 199–215, Mérida, Venezuela. AMSE & ULA. *http://citeseer.nj.nec.com/451592.html.*

[Dávila–Uzcátegui:2000] Dávila, J. A. y Uzcátegui, M. (2000). Galatea: A multi-agent simulation platform. En *International Conference on Modelling, Simulation and Neural Networks [MSNN-2000]*, páginas 217–233, Mérida, Venezuela. AMSE & ULA. *http://citeseer.nj.nec.com/451467.html.*

[Rational:1991] Rational (1991). Rational rose. *http://www.rational.com.*

[Russell–Norvig:1996] Russell, S. J. y Norvig, P. (1996). *Inteligencia Artificial: un enfoque moderno.* Prentice Hall, Inc.

[Shoham:1990] Shoham, Y. (1990). Agent-oriented programming. Technical Report STAN-CS-1335-90, Stanford University, Stanford, CA 94305.

[Uzcátegui:2002] Uzcátegui, M. Y. (2002). Diseño de la plataforma de simulación galatea. Tesis de Maestría, Maestría en Computación, Universidad de Los Andes. Mérida. Venezuela.

[Zeigler:1999] Zeigler, B. P. (1999). Creating simulations in HLA/RTI using the DEVS modelling framework. En *SIW'99*, Tutorial, Orlando, FL.

Glosario

Agente
: Es todo aquellos que puede considerarse que percibe su ambiente mediante sensores y que responde o actúa en tal ambiente por medio de efectores.

Agente Inteligente
: Es aquel que emprende la mejor acción posible en una situación dada. Es un programa autónomo, adaptativo, capaz de aprender de las experiencias y adaptarse a nuevas situaciones. Es algo capaz de percibir y actuar [Russell–Norvig:1996].

Agente Racional
: Es aquel que evita incurrir en contradicciones y si las tuviera intentaría desecharlas de su sistema de creencias. Es aquel que hace lo correcto. Es aquel que, basado en la evidencia obtenido a través de su percepción y el conocimiento que posee, realiza la acción que maximize su medida de rendimiento. Es aquel que realiza la acción correcta [Russell–Norvig:1996].

Agente Reactivo
: Es un agente de bajo nivel, que no dispone de un protocolo ni de un lenguaje de comunicación y cuya única capacidad es responder a estímulos.

Agente Social
: Son aquellos agentes que cooperan con otros agentes.

Arquitectura
: Es un conjunto organizado de elementos, y se utilizan para especificar las decisiones estratégicas acerca de la estructura y funcionalidad del sistema, las colaboraciones entre sus distintos elementos y sus despliegues físicos para cumplir unas responsabilidades bien definidas.

GLIDER

Es un lenguaje de simulación de sistemas en donde cada elemento (o subsistema) del mismo es representado por un ente llamado nodo, un sistema esta compuesto entonces por dos o más nodos los cuales se conectan o comunican formando una red de nodos (grafo orientado o dígrafo) ellos intercambian información unos con otros por medio del pase de mensaje en la red.

HLA (*High Level Architecture*)
: Arquitectura de Alto Nivel. Marco de referencia estándar para el soporte de simulaciones interactivas desarrollado por el DoD.

Inteligencia Artificial Distribuida (IAD)
: Puede ser definida como un campo del conocimiento que estudia e intenta construir conjuntos de entidades autónomas e inteligentes que cooperan para desarrollar un trabajo y se comunican por medio de mecanismos basados en el envío y recepción de mensajes.

Inteligencia Artificial en Paralelo (IAP)
: Esta rama de la IAD se centra en el desarrollo de lenguajes y algoritmos paralelos para sistemas concurrentes en IAD.

La unificación
: Es el proceso de computar las sustituciones apropiadas que permitan determinar si dos expresiones lógicas, ya sean predicados o patrones, coinciden.

Plataforma
: Es un ambiente de software o hardware sobre el que se ejecuta un programa.

POA
: Programación Orientada a Agentes.

POO
: Programación Orientada a Objetos.

RTI (*Run Time Infrastructure*)
: Infraestructura de Tiempo de Ejecución.

Silogismo
: Son esquemas de estructura de argumentos mediante las que siempre se llega a conclusiones correctas si se parte de premisas correctas.

Sistemas MultiAgente (SMA)
: Esta rama de la IAD estudia el comportamiento de agentes inteligentes que resuelven un problema de manera cooperativa.