UNIVERSIDAD DE LOS ANDES

FACULTAD DE INGENIERIA

MAESTRIA EN COMPUTACION

Trabajo de Grado de Maestría

“Integración de la programación declarativa y la programación orientada por objetos para desarrollar agentes inteligentes”

Tutor:

Dr. Jacinto Dávila

Presentado por:

Ing. Jhon Edgar Amaya

Mérida, Febrero de 2002

Integración de la programación declarativa y la programación orientada por objetos para desarrollar agentes inteligentes

Amaya, Jhon Edgar

El proyecto comprende la implementación de mecanismos propios de los lenguajes declarativos, como Prolog, en una arquitectura orientada a objetos como es Java para la programación de Agentes Inteligentes. Se discuten diversas estrategias de integración entre los lenguajes imperativos y declarativos. Se plantea una nueva estrategia de integración consistente en la traducción de los predicados de Prolog a una asociación de clases Java, donde reside un control individualizado declarativo. El control declaractivo individualizado, consiste en la integración de los mecanismos propios de los lenguajes declarativos de Prolog como la resolucion, en cada una de las clases Java vinculadas a los predicados correspondientes. Se describe la arquitectura de agentes inteligentes planteada en [DAV-1] que permite la incorporación de conceptos como racionalidad limitada, metas, influencias, entre otras. Se desarrollaron los programas de traducción automática de la arquitectura de agentes en Prolog a un formato en Java, utilizando la estrategia de integración diseñada.

Palabras claves: Agentes Inteligentes, Java, Prolog, Integración, Racionalidad.

DEDICATORIA

A la vida por la oportunidad del aprendizaje
A mis padres y hermanos por el amor incondicional e irrestricto
A Jacinto y José por su amistad y enseñanzas permanentes

PREÁMBULO

> *"... El software de agentes inteligentes nos ha rodeado desde hace pocos años, aunque esta técnica es aún joven, pero luce prometedora... Las tendencias y desarrollos en el campo de los agentes serán consecuencia de los requerimientos [y necesidades] de los usuarios [entendiendo éstos en su sentido más amplio]... La incertidumbre surge en la pregunta si los usuarios usarán y adoptarán a los agentes por ellos mismos o si ellos comenzarán a utilizarlos porque los agentes serán incorporados [mayoritariamente] a las aplicaciones [como parece ser la tendencia]...." [HER-1]*

Björn Hermann

Este proyecto estudia la integración de la programación declarativa y la programación orientada por objetos para el desarrollo de los agentes de software. La presentación del proyecto se compone de cinco secciones. En la primera sección se presenta una introducción sobre el tema. En la segunda sección se describen las bases conceptuales y demás información necesaria para aclarar los tópicos desarrollados en la Tesis. En la sección tres (3), se desglosan los puntos relacionados con las decisiones tomadas y justificación de diseño planteado, incluyendo la disertación de la metodología empleada. En la sección cuatro (4) se presentan las pruebas realizadas para validar y corroborar la propuesta. En la quinta sección, se discute las conclusiones y recomendaciones correspondientes.

TABLA DE CONTENIDO

TABLA DE CONTENIDO 4

INDICE DE FIGURAS 5

INDICE DE TABLAS 5

INTRODUCCIÓN 5

CAPÍTULO 1. PLANTEAMIENTO DEL PROBLEMA 5

1. OBJETIVOS 5

1.1. Objetivos Generales 5

1.2. Objetivos específicos. 5

2. PROBLEMA GENERAL 5

3. PROBLEMA ESPECÍFICO 5

4. METODOLOGÍA. 5

CAPÍTULO 2. REVISION BIBLIOGRAFICA 5

2. REVISIÓN BIBLIOGRAFICA 5

2.1. LA INTELIGENCIA ARTIFICIAL Y LOS AGENTES 5

2.2. ¿QUE ES UN AGENTE? 5

2.3. CLASIFICACION DE LOS AGENTES 5

2.4. TERMINOLOGÍA EN EL CAMPO DE LOS AGENTES 5

2.5. OBJETOS VS. AGENTES 5

2.6. AGENTES INTELIGENTES 5

2.6.1. AGENTES INTELIGENTES VS. AGENTES TONTOS. 5

2.7. DISEÑO DE AGENTES. 5

2.8. PROLOG 5

2.8.1. IMPLEMENTACIONES DE PROLOG 5

2.8.2. SINTAXIS DE PROLOG 5

2.8.3. RESOLUCIÓN 5

2.8.4. UNIFICACIÓN 5

2.9. JAVA 5

2.9.1. JAVA Y LA PROGRAMACIÓN ORIENTADA A OBJETOS 5

2.9.2. RECURSOS 5

2.9.3. LA INTERFAZ DE USUARIO _____ 5
2.9.4. EVENTOS _____ 5
2.9.5. HEBRAS O HILOS _____ 5
2.9.6. EXCEPCIONES _____ 5
2.9.7. VERSIONES DE JAVA _____ 5
2.10. ¿JAVA Y PROLOG, UNA ALTERNATIVA? _____ 5
2.10.1. JAVA EN PROLOG _____ 5
2.10.2. PROLOG EN JAVA _____ 5
2.10.2.1. MINERVA _____ 5
2.10.2.2. JINNI _____ 5
2.10.3. PROLOG MÁS JAVA _____ 5
2.10.3.1. JASPER. _____ 5
2.10.3.2. INTERPROLOG _____ 5
2.10.3.2.1. *PROGRAMANDO DEL LADO DE JAVA.* _____ 5
2.10.3.2.2. *PROGRAMANDO DEL LADO PROLOG.* _____ 5

CAPÍTULO 3. DESARROLLO CONCEPTUAL _____ 5

3. DESARROLLO CONCEPTUAL _____ 5
3.1. WAM Y LOS MECANISMOS ESENCIALES DE DISEÑO DEL COMPILADOR PROLOG. _____ 5
3.2. TRADUCCION DE PROLOG A JAVA. _____ 5
3.3. DEFINICIÓN DE LOS MECANISMOS DE TRADUCCIÓN AUTOMÁTICA. _____ 5
3.3.1. ESTRUCTURA DE LAS CLASES JAVA ASOCIADAS A PREDICADOS _____ 5
3.3.1.1. IMPLEMENTACIÓN DE LA UNIFICACIÓN DE PROLOG EN LAS CLASES JAVA. _____ 5
3.3.1.2. SIMULANDO LA ESTRATEGIA DE SELECCIÓN DE CLAUSULAS DE PROLOG. _____ 5
3.3.1. DEFINICIÓN DEL CONTROL EN LAS CLASES JAVA. _____ 5
3.3.2. DEFINICION DE LAS CLASES DE CONVERSION A JAVA _____ 5
3.3.3. GENERACION DE LAS CLASES JAVA _____ 5
3.3.4. DEFINICIÓN DE LAS CLASES FUNCIONALES _____ 5
3.3.4.1. CLASE TERM _____ 5
3.3.4.2. CLASE BSCTERM _____ 5
3.3.4.3. CLASE UNIFTERM _____ 5
3.3.4.4. CLASE OPERATORS _____ 5
3.3.5. GENERACION DE CLASE DE CONSULTA _____ 5
3.4. LA PLATAFORMA DEL AGENTE _____ 5
3.4.1. EL AGENTE Y EL MECANISMO DE TRADUCCION _____ 5

3.4.2. GENERADOR DE CLASES ESPECIALES PARA EL AGENTE ____________ 5

CAPÍTULO 4. PRUEBAS Y RESULTADOS ______________________ 5

4. PRUEBAS Y RESULTADOS ____ 5
4.1. PRUEBAS BÁSICAS ____ 5
4.1.1. RESOLUCION Y CONSULTAS. _______________________________________ 5
4.1.2. MULTIMODALIDAD Y NEGACION POR FALLA. _________________________ 5
4.1.3. OPERACIONES ARITMETICAS _______________________________________ 5
4.2. PRUEBAS DEL MOTOR DE INFERENCIA DEL AGENTE ___________________ 5
4.3. BIOINFORMANTES ____ 5
4.4. RESULTADOS ____ 5
4.4.1. MEDICION DEL RENDIMIENTO DE LA PROPUESTA. __________________ 5
4.4.2. MODIFICACION DE LA CLASE DEMOP. ______________________________ 5

CAPÍTULO 5. CONCLUSIONES Y RECOMENDACIONES _________ 5

5.1. CONCLUSIONES ____ 5
5.2. RECOMENDACIONES ____ 5

ANEXOS ____ 5

A. API DEL TRADUCTOR ____________________________________ 5
A.1. Class Term ____ 5
A.2. Class Operators ____ 5
A.3. Class BscTerm ____ 5
A.4. Class ParseTerm ____ 5

B. REGLAS DEL AGENTE DE BIOINFORMANTES. __________________ 5
B.1. Versión 1.0 ____ 5
B.2. Versión 2.0 ____ 5

REFERENCIAS BIBLIOGRAFICAS ____________________________ 5

INDICE DE FIGURAS

FIGURA 2.1. CLASIFICACION DE LOS AGENTES SEGÚN BESSON ___ 5
FIGURA 2.2. CLASIFICACION DE LOS AGENTES SEGÚN FRANKLIN ___ 5
FIGURA 2.3. RELACIONES DE LOS PRINCIPIOS DE DISEÑO DE AGENTES ___ 5
FIGURA 2.4. DESCRIPCION BNF DE PROLOG. ___ 5
FIGURA 2.5. MÁQUINA VIRTUAL DE JAVA ___ 5
FIGURA 2.6. CICLO DE DESARROLLO DE JAVA ___ 5
FIGURA 2.7. HERENCIA Y ATRIBUTOS DE LAS CLASES JAVA ___ 5
FIGURA 2.8. JERARQUÍA DE HERENCIA DE LOS COMPONENTES DE JAVA. ___ 5
FIGURA 2.9. INTERFAZ DE USUARIO DEL APPLET SUMA. ___ 5
FIGURA 2.10. FRONT-END DEL APPLET JINNI ___ 5
FIGURA 2.11. ARQUITECTURA XSB / JAVA. ___ 5
FIGURA 2.12. PROGRAMA JAVA EN INTERPROLOG ___ 5
FIGURA 2.13. SALIDA DEL PROGRAMA INTERPROLOG ___ 5
FIGURA 2.14. SALIDA DEL PROGRAMA PROLOG EN INTERPROLOG ___ 5
FIGURA 3.1. EJEMPLO DE ALMACENAMIENTO DE LA WAM ___ 5
FIGURA 3.2. REPRESENTACIÓN DE UN PREDICADO EN WAM ___ 5
FIGURA 3.3.1. ARBOL DE BUSQUEDA Y METODOS DE CONTROL. ___ 5
FIGURA 3.3.2. ESQUELETO DE LAS CLASES JAVA ASOCIADAS A PREDICADOS ___ 5
FIGURA 3.4. ESQUEMA DE BÚSQUEDA DEL ALGORITMO SLD. ___ 5
FIGURA 3.5. CONTROL DECLARATIVO SIN REGLAS ___ 5
FIGURA 3.6. CONTROL DECLARATIVO CON REGLAS ___ 5
FIGURA 3.7. ÁRBOL DE BUSQUEDA DEL CONTROL DECLARATIVO ___ 5
FIGURA 3.8. CLASES ASOCIADAS AL PROCESO DE PARSING ___ 5
FIGURA 3.9. DIAGRAMA DE LA CLASE SCRIPTW ___ 5
FIGURA 3.10. PROGRAMA GENERADOR DE CLASES ___ 5
FIGURA 3.11. CODIGO DE UNA CLASE CREADA POR GENERADOR CLASES ___ 5
FIGURA 3.12. DIAGRAMA UML DE UNA CLASE CREADA POR GENERADOR CLASES ___ 5
FIGURA 3.13. DIAGRAMA UML DE LA CLASE TERM ___ 5
FIGURA 3.14. DIAGRAMA UML DE LA CLASE BSCTERM. ___ 5
FIGURA 3.15. DIAGRAMA UML DE LA CLASE UNIFTERM ___ 5
FIGURA 3.16. DEFINICION DE OPERADORES EN PROLOG ___ 5
FIGURA 3.17. DIAGRAMA UML DE LA CLASE OPERADOR ___ 5
FIGURA 3.18. CODIGO DE UNA CLASE DE CONSULTA ___ 5
FIGURA 3.19. DIAGRAMA DEL PROGRAMA GENERADOR DE CONSULTAS ___ 5
FIGURA 3.20. PROGRAMA GENERADOR DE CONSULTAS. ___ 5
FIGURA 3.21. CLASE DE CONSULTA ASOCIADA AL PREDICADO NUEVO/3. ___ 5
FIGURA 3.22. DESCRIPCION PICTORICA DEL AGENTE ___ 5

FIGURA 4.1. CONSULTA A UN PROGRAMA DE PRUEBA ____________________________________ 5
FIGURA 4.2. CLASES PARA EL MOTOR DE INFERENCIA____________________________________ 5
FIGURA 4.3. BASE DE CONOCIMIENTO PARA EL PROYECTO DE BIOINFORMANTES _________ 5
FIGURA 4.4. RESTRICCIONES DEL AGENTE DE BIOINFORMANTES _________________________ 5
FIGURA 4.5. DEFINICIONES DEL AGENTE DE BIOINFORMANTES __________________________ 5
FIGURA 4.6. COMPARACIÓN ENTRE PROLOG NATIVO Y LA PROPUESTA EN JAVA BASADO EN EL TIEMPO DE CORRIDA ____ 5

INDICE DE TABLAS

TABLA 2.A. CLASIFICACION DE LOS AGENTES SEGÚN FRANKLIN_________________ 5
TABLA 2.B. PAQUETES PRINCIPALES DE JAVA __________________________________ 5
TABLA 2.C. PROGRAMAS JAVA Y PROLOG _____________________________________ 5
TABLA 3.A. METODOS BÁSICOS DE LA CLASE JAVA______________________________ 5
TABLA 3.B. VECTORES DE CONTROL DE LA CLASE JAVA__________________________ 5
TABLA 3.C. OPERADORES DEFINIDOS EN LA PROPUESTA________________________ 5
TABLA 3.D. DESCRIPCION DE GLORIA____ 5
TABLA 3.E. CLAUSULAS DE PRINCIPALES DE DEMO ____________________________ 5
TABLA 3.F. CLASES GENERADAS POR PROGRAMA TRADUCTOR___________________ 5

INTRODUCCIÓN

> *"Estamos inundados de información pero morimos de inanición de conocimiento" [HER-1]*
>
> John Naisbitt

Este proyecto estudia la integración de la programación declarativa y la programación orientada por objetos para el desarrollo de agentes de software capaces de gestionar el conocimiento en ellos representado.

Poseer datos de algún tipo no es garantía de poseer información o conocimiento alguno. A un ente –bien sea físico o software- que sea capaz de transformar los datos en información o conocimiento le atribuimos inteligencia. En los entornos profesionales de desarrollo de software, que cada vez con más frecuencia enfrentan el desarrollo de sistemas de software complejo, distribuido y en multiplataforma, se hace cada vez más necesario disponer de algunos mecanismos para incorporar mayor "inteligencia" al sistema en cuestión. Uno de los mecanismos más prometedores es lo que se conoce como la tecnología de agentes inteligentes [KNA-1] [WOO-1].

Si bien es cierto que la tecnología de agentes inteligentes, centra su importancia en el ente conocido como agente, no es menos cierto que la potencialidad de dicha tecnología depende en buena medida de la arquitectura del mismo, es decir, en la definición de los elementos del agente y sus interacciones. La arquitectura de un agente establece los mecanismos para gestionar las acciones y procesar las percepciones, de forma tal que el agente cumpla con su cometido.

En [DAV-1] se plantea una arquitectura de un agente que utiliza como base la programación lógica para la incorporación de características tales como, reactividad y apertura al ambiente. Para facilitar el uso de la mencionada arquitectura, en cuanto a la portabilidad, en nuestro proyecto se propone la creación de un software que permita generar una máquina de inferencia en Java a partir de las especificaciones de la mencionada arquitectura.

El concepto de agente se ha convertido en un punto importante en el campo de la inteligencia artificial, ya que ha permitido explorar nuevas soluciones a problemas de gran complejidad. El diseño de agentes representa el estado del arte del desarrollo de componentes de software. El objetivo es incorporar elementos con autonomía,

mecanismos de inferencia, movilidad, así como la comunicación entre dichos elementos; brindando herramientas más eficientes para el desarrollo de software.

No existe un consenso acerca de las pautas para el diseño e implementación de los agentes. En cambio existen diferentes propuestas sobre la teoría computacional que los sustentan y los lenguajes bajo los cuales se programan.

Nuestra propuesta pretende explorar formas alternativas para desarrollar agentes de software. En particular, se estudiará la programación postdeclarativa [WOO-1], en la cual ciertos componentes son programados con orientación a objetos, mientras otros lo son con lenguajes declarativos orientados a lógica.

CAPÍTULO 1. PLANTEAMIENTO DEL PROBLEMA

En el presente capítulo se describe la información acerca de los objetivos planteados por el proyecto, tanto a nivel general como específico. Adicionalmente, se introducen conceptos y disertaciones para bosquejar un panorama acerca del problema que hemos abordado en la Tesis de Grado, incluyendo la metodología propuesta para la consecución de los objetivos.

1. OBJETIVOS

1.1. Objetivos Generales

Definir la intercepción de la programación lógica y la orientación por objetos para el desarrollo de agentes, con respecto a la arquitectura de agente inteligente descrita en [DAV-1].

Discutir la conveniencia de los paradigmas declarativo, imperativo y la orientación por objetos en cada etapa y componentes de desarrollo de software para agentes inteligentes.

Evaluar la factibilidad de “especificaciones ejecutables” para agentes de software.

1.2. Objetivos específicos.

Desarrollar un traductor de las especificaciones lógicas de la arquitectura [DAV-1] a un lenguaje orientado a objeto.

Desarrollar procedimientos de traducción sistemática o automática de especificaciones lógicas de agentes (en Prolog) en código orientado por objetos (en Java).

Generar motores de inferencia alternativos en lenguaje Java, usando el traductor diseñado para interpretar el código lógico.

Evaluar los procedimientos de traducción con ejemplos concretos.

2. PROBLEMA GENERAL

La Inteligencia Artificial ha evolucionado desde sus primeros pasos, en los que la intención por extrapolar la capacidad de resolución de problemas que poseía el hombre a un dispositivo, denotando la presencia de "inteligencia", era su fin último. Actualmente, el objetivo global es la creación de entes que pueda actuar de manera "racional" y autónoma, y de ser posible, en diferentes ambientes. [KNA-1]

A esos entes se les conoce como agentes inteligentes y les caracteriza su autonomía, sus estados internos, la definición de metas y el poder de decisión [BIG-1].

> *"Los más difícil del diseño de agentes inteligentes es colocarles la i [de inteligentes]".*

Michael Knapik

La autonomía está caracterizada por la capacidad que tiene el agente para responder a los cambios circundantes, sin la necesidad de la intervención del programador. [KNA-1] define el concepto de autonomía como:

> *"La autonomía consiste en la capacidad de los agentes de operar sin la intervención de los humanos u otros entes, y [que el agente] tenga algún tipo de control sobre su estado interno".*

Michael Knapik

Las naves de exploración lanzadas al espacio (Pathfinder, por ejemplo), deben tener una gran autonomía, debido a que el tiempo para establecer la comunicación entre el agente y la estación terrena es enorme para efectos de control. El tiempo que se requeriría para que dicho objeto enviase una señal y se procesara la información correspondiente en la Tierra, y luego se enviara la respuesta, podría llevar de unos cuanto minutos a quizás horas.

Bajo este ejemplo, se deduce que éste "agente", no sólo debe reaccionar al ambiente inhóspito y en cierta medida desconocido. Debe ser capaz de manipular conocimiento y aprender de los datos adquiridos por sus sensores y tomar las decisiones que mejor se adapten a la situación dada.

Esta manipulación de conocimiento debe ser lograda a través de la definición de estados internos propios de cada agente, así como la definición de reglas que marquen las metas que persigue el agente en cuestión. Pero adicionalmente, debe definirse una plataforma a través de la cual el agente puede administrar el conocimiento. A este compendio se le denomina arquitectura.

La arquitectura incluye los mecanismos de razonamiento, el tratamiento de los datos de entrada, así como la base de datos de conocimiento y los medios para su manipulación, y adicionalmente, establece cómo se realiza el proceso de toma de decisiones.

El gran problema del diseño de agentes inteligentes es la dificultad de incorporar los elementos necesarios para el raciocinio [KNA-1]. En [DAV-1] se plantea una arquitectura que utiliza la programación lógica para la especificación de agentes que integren características tales como, reactividad, apertura, entre otras.

Entre los primeros lenguajes utilizados para la programación de agentes inteligentes se cuenta la programación lógica [RUS-1]. La programación lógica se fundamenta en dos principios básicos:

> *"[La programación lógica] utiliza lógica para expresar el conocimiento y utiliza inferencia para manipular el conocimiento". [HOG-1]*

Christopher Hogger

El lenguaje de programación lógica por excelencia es el Prolog. El Prolog utiliza un subconjunto particular de la lógica de predicados de primer orden para la representación del conocimiento y utiliza una regla de inferencia denominada resolución para la manipulación de dicho conocimiento. La idea general de la programación lógica, consiste en inferir desde un contexto particular y unas suposiciones dadas, una conclusión deseada [RUS-1].

En Prolog hay una disciplinada separación del conocimiento y de cómo se utiliza. De forma tal que el código es declarativo y, en general, no determinístico.

Muchos compiladores modernos de Prolog están basados en descripción de implementación conocida coma la máquina abstracta de Warren o mejor conocida como WAM, la cual es una especificación de una máquina virtual que permite optimizar los traductores para la plataforma donde se realiza la operación de compilación.

En cierta forma, esto es muy cercano a la estrategia Java de compilar programas para una máquina virtual. En Java existe la posibilidad de ejecutar programas en múltiples plataformas sin necesidad de compilarlo para cada una de ellas. Los códigos fuentes de Java son compilados en un código intermedio –para la máquina virtual- que puede ser ejecutado dondequiera que se disponga de una máquina virtual Java. Este código intermedio no es específico de cada plataforma, pero si lo es la implementación de la máquina virtual Java. Adicionalmente Java incluye bibliotecas, manejo de redes y seguridad, GUIs[1], manejo de eventos, multiprogramación, entre otras. Según [NAU-1], Java presenta las siguientes características generales:

> *"simple, seguro, portable, orientado a objeto, robusto, multihebra, arquitectura neutral, distribuido, dinámico, de alto performance".*

Patrick Naughton

Según [KNA-1], la programación de agentes utilizando lenguajes orientados a objetos presentan los siguientes beneficios:

> *"código reusable, reducción de costos de desarrollo, estructura flexible, extensible, conexión jerárquica de agentes y dominios".*

Michael Knapik

En la programación de agentes que utilizan lenguajes orientados a objetos, incluyendo Java, el conocimiento y control se plasman juntos en el mismo código. Una ventaja muy importante de la programación de agentes en Java consiste en la posibilidad de implementar un sistema multiagentes en la Internet [LAN-1]

De la totalidad de disertación anterior, surgen una serie de preguntas o interrogantes en el diseño de agentes, que forman parte del marco de motivaciones del presente proyecto: ¿Cómo debe programarse un agente para cumplir con las condiciones de básicas de funcionalidad? ¿Cómo se integran propiedades ortogonales de los agentes, tales como movilidad, sociabilidad, creencias, orientación a metas, entre otras? ¿De ser así, cómo deben codificarse? ¿Cuál arquitectura de agentes es la que proporciona mayor flexibilidad, mejores prestaciones para el cumplimiento de sus metas, mayores posibilidades para la representación del conocimiento y su consecuente manipulación?.

[1] Acrónimo de Interfaz Gráfica de usuario (en inglés Graphic User Interface)

Wooldridge [WOO-5] hace una disertación de diferentes arquitecturas planteando las ventajas y desventajas de cada una de ellas y responde otras preguntas como: ¿Cuál es el lenguaje que facilita la programación de agentes? ¿Basta un solo lenguaje para la programación de agentes? ¿Se puede combinar varios lenguajes de programación para lograr una fórmula más efectiva para la programación de agentes inteligentes?. [BIN-1] considera la integración de dos lenguajes como Java y Prolog, como una excelente combinación, ya que garantiza flexibilidad –que brinda Java- y expresividad –que da Prolog- a la hora de la programación de aplicaciones.

En las siguientes secciones, se abordará un subconjunto de estas últimas interrogantes, y se planteará una estrategia para la solución de ese escenario de investigación.

3. PROBLEMA ESPECÍFICO

En el diseño de agentes es necesaria la definición de la teoría, arquitectura y lenguaje que lo implementarán. Existen diferentes propuestas de cómo debe implementarse un agente [KNA-1][WOO-3][KOW-1]. Wooldridge plantea algunas alternativas en cuanto al manejo de estas propuestas [WOO-5].

Dentro de la teoría de agentes existe una que se fundamenta en la extrapolación del comportamiento humano, utilizando para predecir y modelar éste comportamiento, la definición de atributos denominados actitudes, tales como creencias, deseos, esperanzas, etc., basados en la teoría psicológica de Dennett [WOO-5]. La intención es definir estructuras que tomen decisiones.

[KOW-1] propone la creación de una arquitectura de agentes que se sustente en la programación lógica, y que permita la reactividad junto con la racionalidad del agente, según se cita a continuación.

> *"Se propone una arquitectura unificada... entre las arquitecturas de agentes racionales y reactivos [híbrida]... para tal propósito se propone emplear un procedimiento de prueba como el componente de pensamiento del agente, incluyendo el procedimiento de prueba definiciones e integridad de restricciones... Esta arquitectura utiliza la programación lógica para la reducción de metas a submetas de forma racional... y [para la reactividad se] utilizan reglas de condición-acción para conservar integridad de restricciones...[Ya que el agente no puede pensar por siempre] se emplea la formalización de recursos limitados en el procedimiento de prueba, así el agente puede interrumpir y resume el proceso de pensamiento, con la intención para grabar y asimilar las observaciones como una entrada y ejecutar acciones atómicas como salida"*

Robert Kowalski

Dávila plantea una arquitectura de un agente inteligente que utiliza la programación lógica de forma generalizada para la programación de agentes que integren características tales como, reactividad, apertura, así como las nociones de creencias, objetivos y actividades mentales [DAV-1]. Podemos observar que tenemos dos componentes básicos planteados por Kowalski y Wooldridge, es decir una arquitectura híbrida con una visión antropomórfica en la toma de decisiones.

La arquitectura propuesta por Dávila fue totalmente realizada en Prolog. La idea ahora es ampliar el espectro de posibles aplicaciones de la arquitectura, utilizando como complemento o soporte un lenguaje orientado a objeto como Java. Calejo [CAL-2] nos proporciona una discusión interesante acerca del tema.

> *"Los programadores en lógica necesitan un segundo lenguaje; el mundo simplemente no es totalmente declarativo. A menos que se esté dispuesto a ignorar esto [ultimo], o apegarse a un ambiente Prolog altamente pesado para proporcionar todas las operaciones primitivas de OS/GUI/DB... es obvio [entonces] que se requiere un segundo lenguaje para realizar proyectos más completos y complejos. Esto se debe a tanto a razones técnicas como económicas. Los programadores en lógica debemos dejar la puerta abierta para la innovación (es decir, ampliando procedimientos de prueba, lenguajes de mayor nivel, etc.); y el costo de integración y mantenimiento son un factor para la escogencia de las herramientas. En otras palabras, el trabajo en ambientes Prolog (quizás) es mucho mejor si se enfoca en las máquinas declarativas, delegando otros problemas a otra parte..., Java emerge [como esa parte restante] como el lenguaje a escoger para muchos de los contexto del mundo real [las interfaces con el usuario], constituyendo una excelente plataforma para este escenario... por su amplio uso, capacidad multiplataforma, orientado a objeto y dinámico"*

Miguel Calejo

Dado que las implementaciones de Java y Prolog difieren considerablemente (como se discute en las secciones 3.8, 3.9 y 3.10), es necesario definir algún método para la realización de este proceso de traducción, para conservar la integridad del proceso. [TAR-1], [WOO-2], [WOO-4]. La idea implícita de la traducción de aplicaciones en lenguajes lógicos a orientados a objeto, es dotar al diseñador de agentes de mecanismos de resolución e inferencia, aprovechando además las ventajas propias de los lenguajes orientados a objeto. En [COD-1] se exploran diferentes posibilidades para la traducción de Prolog a lenguaje C, definiendo una amplia gama de estructuras de datos para realizar la mencionada tarea, junto con la discusión de las ventajas y desventajas inherentes de cada una de ellas.

Sobre la arquitectura representada por Dávila, se pretende construir una plataforma que permita traducir de manera sistemática la lógica de predicados a clases Java. Los traductores serán automáticos en la medida de lo posible y se les especializará en la traducción de especificaciones de **agentes**.

Nuestro objetivo inmediato es ofrecer las traducciones Java de los agentes para que sean empleados en otros ambientes, en particular en la plataforma de simulación de sistemas multiagentes GALATEA u otros proyectos específicos.

En conclusión podemos reafirmar que nuestra propuesta de Tesis tiene por objetivo evaluar paradigmas alternativos y complementarios para desarrollar agentes inteligentes de software. Los paradigmas a considerar son el diseño orientado a objeto [BIG-1] y la programación lógica [RUS-1] que, como hemos argumentado, ofrecen servicios complementarios al desarrollo del software inteligente. Para cumplir con el objetivo planteado se desarrolla un software que permite traducir las especificaciones lógicas de la arquitectura planteada por [DAV-1], de forma tal de generar los motores de inferencia para los agentes en lenguaje Java.

4. METODOLOGÍA.

A continuación se describe de forma abreviada los pasos metodológicos seguidos para la consecución del proyecto de Tesis. Recalcamos el término abreviado, ya que en el capítulo tres se desarrolla con mayor especificidad tanto los detalles de diseño como la metodología propiamente dicha.

El proyecto se dividió en cinco etapas bien definidas, y que se describen a continuación:

1. Se realizó un estudio acerca de los métodos de traducción de Prolog a Java, y se determinó cuál de ellos se adapta mejor a nuestros fines, basándonos en [LEV-1][COD-1].

Existen diferentes métodos o formas para implementar las versatilidades de Prolog en Java, como se discutirá en el capítulo 2, ya que la máquina virtual de Java es diferente a la implementación de la máquina abstracta de Warren. En [COD-1] se plantean formas de implementar Prolog en algunos lenguajes de programación. En [JPR-1][COD-1] se establece algunos métodos para "traducir" código en Prolog a código en lenguaje C.

Luego, de estudiar los diferentes métodos se determinó cual de ellos se adapta mejor a nuestro objetivo. [SIC-1][BIN-1]

2. Se generó un programa que convierta las especificaciones en Prolog a un código Java. [JPR-1] [CAL-1][TAR-1][TAR-2].

El objetivo de esta etapa fue la creación de una aplicación que genere de forma automática el código Java a partir de las especificaciones dadas en Prolog. [WOO-3][WOO-6] plantea algunas pautas para conservar la integridad de los objetos creados para la programación de un agente.

3. Se implementó las especificaciones lógicas de la arquitectura [DAV-1] en el traductor.

En esta etapa se procedió a implementar diferentes estructuras planteadas en [DAV-1], que serán discutidas en el capítulo 3. La intención de esta etapa fue verificar el buen funcionamiento del "traductor".

4. Se generó un motor de inferencia en Java para un agente, a partir de las especificaciones lógicas de la arquitectura.

5. Se estableció el rendimiento del motor de inferencia en Java. Para ello se implementó un ejemplo de un agente y se procedió a realizar mediciones del tiempo para el cumplimiento de los objetivos de dicho agente.

La intención fue medir el rendimiento de la solución propuesta, para ello implementamos un máquina de inferencia en Prolog, ya existente y planteada por Dávila, y la contrastamos con la solución dada por nuestra propuesta bajo Java.

En el capítulo sobre el desarrollo conceptual de nuestro proyecto, se desglosará con detalle cada uno de los pasos dados, así como los métodos utilizados para la consecución del software.

CAPÍTULO 2. REVISION BIBLIOGRAFICA

2. REVISIÓN BIBLIOGRAFICA

2.1. LA INTELIGENCIA ARTIFICIAL Y LOS AGENTES

Es indudable que la concepción de inteligencia artificial ha sido un tema de profundo interés para los investigadores de este campo, desde sus albores en la década de los años 40, con la idea de construir dispositivos tan inteligentes como los seres humanos.

> *"... la interesante tarea de lograr que las computadoras piensen… máquinas con mente, en su sentido literal" [RUS-1]*

Haugeland

Algunos investigadores difirieron –y difieren- de esta concepción, por considerarla demasiado "antropomórfica", es decir, el fin último de la IA[2] estaría vinculado a la definición y construcción de máquinas que fuesen similares al hombre. La discusión se centra en si el ser humano debe ser tomado como el mejor modelo de inteligencia, a sabiendas que bajo ciertas circunstancias sus decisiones pueden ser erróneas. La pregunta que surge es qué otro modelo puede ser tomado para definir lo que es IA.

Esa otra corriente filosófica [RUS-1] sobre la conceptualización de la IA, afirma que no necesariamente un ente tiene que actuar o comportarse como un ser humano para ser un ente inteligente. Propugna por un modelo en el cual la inteligencia de un determinado ente sea valorada por el grado de razonamiento del mismo. Shalkoff, pertenece a esta corriente de pensamiento y opina que la IA:

> *"...se enfoca en la explicación y la emulación de la conducta inteligente en función de procesos computacionales " [RUS-1]*

Schalkoff

Determinar la asertividad de alguna de las definiciones de Inteligencia Artificial, siempre involucrará la inherente discusión filosófica de la definición de inteligencia. Los investigadores prefieren omitir la discusión sobre el concepto de inteligencia y se

[2] Acrónimo de Inteligencia Artificial

concentran en la creación de entes –que en adelante denominaremos agentes- que pueda actuar o razonar inteligentemente.

Un ***agente*** puede definirse a grosso modo como un ente que puede percibir su ambiente circundante mediante sensores y que responde o actúa sobre éste por medio de efectores. Una definición más completa de agente es dada por [DAV-1]

> *"...es una entidad que percibe su ambiente, puede asimilar dichas percepciones incorporándolas en dispositivos de memoria, puede razonar con la información en estos dispositivos, puede adoptar creencias, objetivos e intenciones por si mismo y puede procurar activamente dichas intenciones, a través del control apropiado a sus efectores" [DAV-1]*

Jacinto Dávila

Dentro de este concepto, existe inherentemente la inclusión de racionalidad así como definición de características vinculadas al ser humano. Ahora bien, la disyuntiva obligatoria consiste en delimitar lo que es o no racional. Se acepta que:

> *"...un agente perfectamente racional actúa en todo momento de manera tal que logra maximizar su utilidad esperada, con base en la información que ha obtenido del entorno" [RUS-1]*

Stuart Russell

El logro de la racionalidad perfecta, es decir, hacer siempre aquello que es correcto, es muy difícil de conseguir en entornos complejos debido al alto precio en cuanto a los requerimientos de cómputo. El diseñador de agentes inteligentes -y por lo tanto racionales-, debe convenir con el establecimiento de límites o espacios acotados, es decir, el agente se comporta lo mejor que puede dentro de lo que permiten los recursos de cómputo asociados a él. Esta propiedad se conoce como racionalidad limitada.

La "creación" de agentes ha sido en años recientes la preocupación de la inteligencia artificial, y consecuentemente la meta de la investigación de algoritmos y métodos de programación de éstos. La programación de agentes es llevada a cabo en diferentes lenguajes dependiendo de las facilidades computacionales en las cuales pueda ser implementado dicho agente.

2.2. ¿QUE ES UN AGENTE?

Existen varias definiciones acerca de lo que es un agente, dependiendo de los diferentes puntos de vista utilizados, debido tal vez a que el término es usado por muchas personas que trabajan en áreas afines.

En una definición primigenia de lo que es un agente, éste puede concibirse como "*Alguna cosa que produce o es capaz de producir un efecto, o alguien que actúa en lugar de otro por autoridad de él, o un medio o instrumento por el cual una guía inteligente alcanza un resultado*", pero que no conllevan al espíritu del contexto y razón de ser de agente, ya que carecen de algunos elementos necesarios para definir su agencia.

Russell et al. [RUS-1] define **agente** de tres formas distintas, dependiendo del grado de sofisticación o inteligencia:

a. En forma genérica y conceptual: "*Un agente es cualquier cosa que pueda ser vista como perceptores de su ambiente a través de sensores y actuando sobre el ambiente por medio de efectores".*

b. Con la adición de racionalidad: "*Para cada posible secuencia de percepción, un agente racional ideal, haría cualquier cosa para maximizar su medida de rendimiento, en base a la evidencia proporcionada por la secuencia de percepción y cualquier conocimiento que el agente tenga*"

c. Inteligencia como racionalidad y autonomía: "*un agente es autónomo si puede extender sus acciones o escogencias dependiendo de su propia experiencia, en lugar del conocimiento del ambiente que ha sido construido por el diseñador*"

Una definición bastante general de lo que es un agente es dada por Atkinson et al.:

> *"Los agentes inteligentes son entidades de software que transportan algún conjunto de operaciones para provecho de un usuario u otro programa con algún grado de independencia o autonomía, y al hacer esto emplear algún conocimiento o representación de los deseos y metas de los usuarios. Los agentes inteligentes entonces pueden ser descritos en términos de un espacio definido por estas dos dimensiones de agencia e inteligencia.*

La agencia es el grado de autonomía y autoridad conferidas al agente y puede ser medido al menos cualitativamente, por la naturaleza de las interacciones entre el agente y otras entidades en el sistema. Como mínimo un agente debe correr asincrónicamente. El grado de agencia aumenta si un agente representa de alguna forma a un usuario. Esto último es uno de los valores claves de los agentes. Un agente más avanzado puede interactuar con otras entidades tales como datos, aplicaciones o servicios. Agentes más avanzados colaboran y negocian con otros agentes.

La inteligencia es el grado de razonamiento y aprendizaje: La habilidad del agente de aceptar la declaración de metas de los usuarios y transportar las tareas delegadas a éste. Como mínimo puede haber algunas declaraciones de preferencias, quizás en la forma de reglas con una máquina de inferencia o algún otro mecanismo para actuar sobre estas preferencias. Niveles más altos de inteligencia incluyen un modelo de usuario o alguna otra forma de entendimiento/razonamiento acerca de lo que el usuario quiere hacer. Más allá de la escala de inteligencia éstos, son sistemas que aprenden y se adaptan a su ambiente, ambos en términos de los objetivos del usuario, y en términos de los recursos disponibles del agente. Un sistema puede, como un asistente humano descubrir nuevas relaciones, conexiones o conceptos independientemente del usuario humano, y explotar esta anticipación y satisfacer las necesidades del usuario". [ATK-1]

Una definición operacional de lo que es un agente estará compuesta de los siguientes aspectos, que están vinculados profundamente con los atributos del mismo [WOO-5][KNA-1]:

a. **autonomía**: Los agentes operan sin la intervención directa de los humanos u otros entes, y tienen algún tipo de control sobre su estado interno.

b. **Habilidad social**: Los agentes interactúan con otros agentes (y posiblemente humanos), vía algún tipo de lenguaje de comunicación entre agentes.

c. **Reactividad**: Los agentes perciben su ambiente y responden en una fracción de tiempo a los cambios que ocurren en éste.

d. **Proactividad**: Los agentes no simplemente actúan en respuesta a su ambiente, ellos son capaces de exhibir comportamientos dirigidos a metas con el fin de tomar la iniciativa.

Existen propiedades ortogonales de la agencia, es decir, que pueden presentarse en los agentes pero no son necesariamente una característica esencial para ser considerados agentes, como por ejemplo, movilidad, veracidad, benevolencia, racionalidad, entre otras. [LAN-1]

Los agentes también suelen ser definidos en términos del dominio en los cuales ellos proporcionan sus servicios, incluyendo:

a. Búsqueda de información
b. Filtraje de datos
c. Proveer acceso y seguridad transaccional
d. Siendo actores o actrices en películas
e. Optimizando técnicas orientadas a metas, entre otras.

A medida que la técnica computacional avanza, el advenimiento de sistemas heterogéneos distribuidos más complejos y el uso de redes, significa que el desarrollo de software usando métodos de programación tradicional se han vuelto cada vez más difíciles, por lo que han incorporado o utilizado agentes dentro de esta nuevas aplicaciones de sistemas distribuidos, volviéndose cada vez más atractivo este concepto [KNA-1][HER-1]. Por ejemplo, los usuarios que intentan indagar manualmente a través de cientos y miles de páginas en la Internet, y otras fuentes de información, lo que implica que el usuario sufre algún tipo de frustración debido a lo azaroso que puede resultar la búsqueda. Sin embargo, los agentes emergen como localizadores analizadores e indagadores de información, por lo tanto los usuarios harán su búsqueda mucho más efectiva. [HER-1]

Una pregunta que surge de manera natural es qué conceptos específicos dentro de la inteligencia artificial participan en la generación de inteligencia en los agentes. A continuación una breve discusión sobre la posible respuesta a esta pregunta.

Muchos de los sistemas basados en agentes involucran la existencia, creación y/o adquisición de conocimiento. El conocimiento es alguna cosa, por ejemplo, un dominio, el inventario de un supermercado, etc. El uso de este conocimiento involucra las formas en las cuales los agentes pueden representar su mundo – generalmente este concepto se denota como base de conocimiento - y varios tipos de lógica - tales como booleana, proposicional, cálculo de predicado de primer orden, modal, difusa- que pueden ser utilizadas para realizar la inferencia en la base de conocimiento.

La inferencia o el razonamiento sobre la base de conocimiento conduce a la generación de creencias. La capacidad de razonamiento es importante para que el agente trate de

verificar el valor de verdad de sus propias creencias, o aquellas hechas por otros agentes; por lo tanto, la inferencia puede ser utilizada para determinar una secuencia de acciones necesaria para alcanzar una meta.

Algo definido como verdad por un agente en su mundo, - definido en su base de conocimiento -, puede no ser transferido o no a otros dominios o contextos y mantenerse consistencia de su base de conocimiento. Algún asunto de transformación o calificación podría ser considerado. Esto último es importante cuando se convienen con sistemas de agentes distribuidos, donde un agente puede encontrarse en un ambiente diferente desde el cual fue inicializada la base de conocimiento de éste.

Podría concluirse entonces que una base de conocimiento es una colección de conceptos acerca de un dominio o dominios, y por lo tanto, definir a la ingeniería de conocimiento como el proceso por medio del cual aquellos conceptos son extraídos codificados y relacionados con otros. La forma en la cual el conocimiento es realizado y el vocabulario utilizado para definir los conceptos del domino, representan una teoría o un modelo de existencia acerca de este dominio

En [FRA-1] se plantea una consideración sobre como la percepción humana de inteligencia causa una influencia en la definición de agente:

> *``Nos ayuda a entender el porqué el llegar a una definición definitiva del concepto de agente es tan difícil: lo que para una persona es un agente inteligente para otra es un objeto "listo"; y lo que hoy es un objeto "listo", mañana será un programa inútil. La clave de esta distinción está en nuestras expectativas y en nuestro punto de vista"*

La idea de agente se ve fortalecida si se toma una visión antropomórfica de éstos, es decir entidades con sensaciones, percepciones y emociones como los seres humanos [WOO-5].

Una discusión sobre las características de lo que es un agente y los diferentes atributos que lo definen, puede ser vista en [FRA-1].

2.3. CLASIFICACION DE LOS AGENTES

No existe un consenso sobre la clasificación de los agentes, existen diferentes esquemas que muestran potenciales sistemas para la clasificación de los agentes, dependiendo del ámbito del autor de este esquema.

Un esquema de clasificación de agentes desarrollado por [BES-1], está orientado a una concepción desde el punto de vista de la robótica. En la figura 2.1, se muestra dicha clasificación.

[FRA-1] propone una clasificación más orientado al área de la ingeniera del software, centrándose en los agentes de software. En la figura 2.2, se muestra la propuesta de clasificación de Franklin.

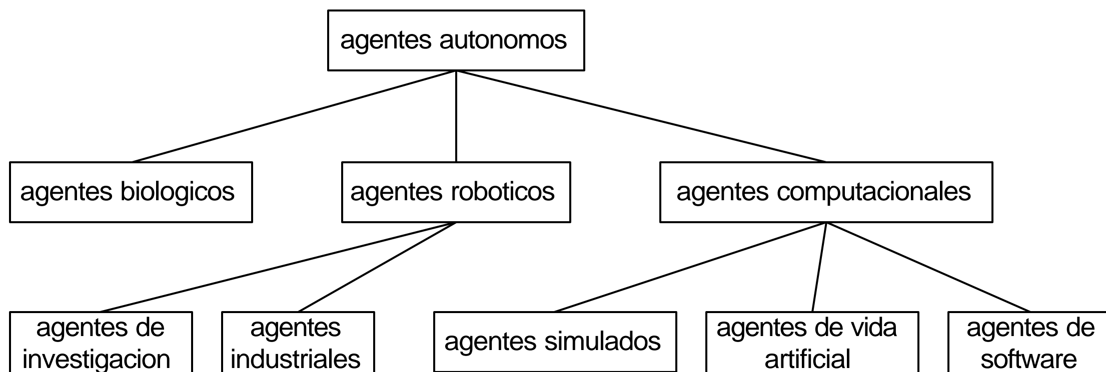

FIGURA 2.1. CLASIFICACION DE LOS AGENTES SEGÚN BESSON

Es importante notar que en estos sistemas de clasificación tienen como punto de partida la autonomía del agente, es decir, se considera una condición mínima necesaria para la agencia.

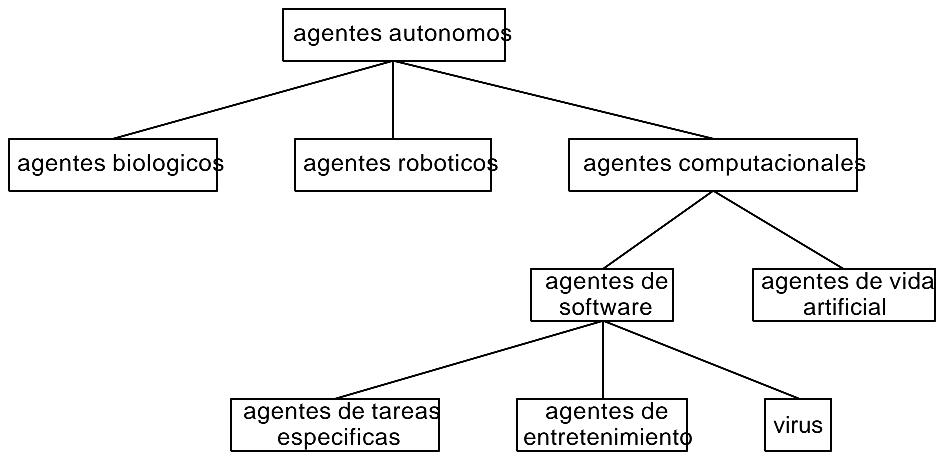

FIGURA 2.2. CLASIFICACION DE LOS AGENTES SEGÚN FRANKLIN

Las clasificaciones presentadas están orientadas a agrupar a los agentes en grades grupos, pero sin tomar en cuenta las especificaciones vinculadas con los atributos que nos permite definir la agencia. [FRA-1] plantea un esquema de clasificación que define al agente de acuerdo a la propiedad que presenta, sin ser mutuamente exclusivas unas de otras. En la tabla 2.a., se muestra dicha clasificación.

Propiedad	Otro nombre	Significado
Reactividad	**(sensar y actuar)**	**Responder en una fracción de tiempo a los cambios presentes en el ambiente**
Autonomía		**Ejerce el control sobre sus propias acciones**
Orientado a meta	**Proactividad Orientado a propuesta**	**No actúa simplemente en respuesta al ambiente**
Continuidad temporal		**Es un proceso que corre continuamente**
Comunicación	**Capacidad social**	**Se comunica con otros agentes, quizás incluyendo personas**
Aprendizaje	**Adaptabilidad**	**Cambia su conducta basado en las experiencias previas**
Movilidad		**Capacidad para transportarse desde una máquina a otra**
Flexibilidad		**Las acciones no esta escritas**
Caracterización		**Creencia en una "personalidad" y en estado emocional**

TABLA 2.A. CLASIFICACION DE LOS AGENTES SEGÚN FRANKLIN

2.4. TERMINOLOGÍA EN EL CAMPO DE LOS AGENTES

En el campo de la investigación de agentes se ha adoptado algunos términos [FLO-1] para referirse al contexto, agrupaciones, entre otros. A continuación algunos términos importantes:

a. **Teoría de agentes:** Para Wooldridge, representa la definición de la agencia, así como el uso de formalismos matemáticos para la representación y razonamiento de las propiedades de los agentes [WOO-5]

b. **La arquitectura del agente:** Las arquitecturas de agentes ven a los agentes como un conjunto de componentes de percepción, acción y razonamiento. Los componentes de percepción alimentan los de razonamiento, los cuales gobiernan las acciones de los agentes, incluyendo lo que se va a percibir a continuación. [FLO-1]

   Wooldridge [WOO-5] partiendo de la concepción clásica de IA, es decir la representación simbólica de algún tipo de conocimiento, plantea tres tipos de arquitectura:

   Declarativa: Es aquella que contiene "una representación explicita el modelo simbólico del mundo, y en las cuales las decisiones son hechas a través de razonamiento lógico o pseudológico, basado en la coincidencia de patrones (matching) y la manipulación simbólica". [WOO-5]

   Reactiva: Dado que algunas observaciones han demostrado que algunos problemas son más complejos de solucionar con las técnicas simbólicas clásicas. Wooldridge la define como la antitesis de la arquitectura declarativa, es decir, sin ningún modelo simbólico del mundo.

   Híbrida: Esta arquitectura tiene por objetivo la suma de lo mejor de ambos mundos (deliberativos y reactivos). Agentes con capacidad de planificar y tomar decisiones según la concepción del mundo a través de su modelo simbólico, así como, ser capaz de reaccionar a eventos de una forma inmediata[3], sin estar atado a un excesivo razonamiento.

[3] La inmediatez a la que se hace referencia constituye un tiempo mínimo de actuación, es decir, tan pronto como se pueda.

c. **La arquitectura de sistemas de agentes:** Analiza los agentes como entidades interactivas que proporcionan y consumen servicios. Las arquitecturas de sistemas facilitan las operaciones y las interacciones de los agentes bajo las restricciones del entorno, y les permiten aprovechar las facilidades y los servicios disponibles.

d. **El marco de trabajo[4] del agente[5]:** Representa la herramienta de programación para la construcción de agentes. Un ejemplo de éste, es el sistema de *aglets*. [LAN-1]

e. **Infraestructura de agentes:** Proporcionan las reglas que los agentes utilizan para comunicar y entenderse entre ellos.

En [FLO-1] se discute de forma muy elegante la diferencia de Infraestructura y Arquitectura de agentes.

2.5. OBJETOS VS. AGENTES

Los agentes y los objetos comparten muchas características; cosa que a veces hace difícil el poder diferenciarlos. Por ejemplo, la programación orientada a agentes o AOP[6] podría ser considerada como una instancia del paradigma de la programación orientada a objetos u OOP[7]. La OOP ve los sistemas como un conjunto de objetos comunicándose entre ellos para realizar operaciones internas, mientras que AOP se especializa en ver agentes (en lugar de objetos), cuyas operaciones internas se basan en creencias, capacidades y elecciones, que se comunican con otros usando mensajes [BIG-1][FLO-1]. La idea de la programación orientada a agentes fue propuesta por Yoav Shoham [SHO-1], basándose en un punto de vista social. La idea central consiste en diseñar los programas como sociedades de agentes que interactúan entre ellos.

[FLO-1], plantea tres vertientes básicas que diferencian a los agentes de los objetos.

[4] Framework por su término en inglés.

[5] Otro autores –Wooldridge por ejemplo- prefieren el término *lenguaje de agente*, al software que permite la programación y experimentación con agentes.

[6] Agent Oriented Programming por su nombre en inglés

[7] Object-Oriented Programming por su nombre en inglés

La primera es el grado en el que los agentes y los objetos son autónomos. No pensamos en los agentes como invocadores de métodos entre ellos, sino pidiendo la realización de acciones. En el caso orientado a objetos, la decisión recae en el objeto que invoca el método. En el caso de los agentes, la decisión recae en el agente que recibe la petición.

La segunda distinción importante... es respecto a la noción de un comportamiento autónomo y flexible (reactivo, preactivo, social)

La tercera dstinción importante... es que se considera que cada agente tiene su propio flujo de control... en el modelo de objetos estándar, hay un único flujo de control para todo el sistema.

De la anterior discusión podemos observar como los agentes cumplen con las características propias de la agencia [KNA-1], en especial la autonomía. Bigus lo resume en la siguiente frase:

"Los objetos son paquetes de datos a la espera de que un botón sea pulsado. Los agentes deciden activamente que botón pulsarán [o que hacer cuando el botón es pulsado]" [BIG-1]

2.6. AGENTES INTELIGENTES

Los agentes inteligentes son vistos como entidades que emulan procesos mentales o simulan un comportamiento racional [BIG-1].

La inteligencia de un agente está relacionada con la capacidad de actuar racionalmente bajo las circunstancias captadas por el agente. La racionalidad se refiere al proceso de escoger la acción óptima, dada una cantidad de información conocida por el agente. Kowalski plantea una de las principales dificultades de la agencia racional

"Un base de datos o base de conocimiento tradicional contienen información simbólica, frecuentemente en forma lógica. Una base de conocimiento puede recibir entradas, las cuales son actualizaciones o preguntas. [El agente] puede chequear si las actualizaciones satisfacen las integridades de las restricciones. Sin embargo, solamente la salida puede generarse en forma de respuesta a las preguntas [uno de

los tipos de entrada]. La cantidad de recursos que ésta puede necesitar para deducir tales respuestas puede ser ilimitado". [KOW-1]

Como se puede deducir de la cita anterior, la decisión acerca de cuál es la acción óptima involucra el conocimiento de todas las variables del ambiente y el análisis de todos los posibles escenarios que involucre la mejor acción, una suerte de predictor del futuro. Este mecanismo involucra una gran cantidad de tiempo así como posiblemente una cantidad infinita de almacenamiento.

El proyecto DeepBlue, en el cual el computador puede analizar cerca de 1.000.000 de posibilidades o escenarios por segundo, con una profundidad de 8 a 9 jugadas[8]. Algunos sistemas no pueden poseer toda la información para tomar la decisión óptima, en este caso se toma la mejor decisión que la ocasión amerite, debido a las restricciones de tiempo y espacio computacional, o conocido como racionalidad limita o acotada.

2.6.1. AGENTES INTELIGENTES VS. AGENTES TONTOS.

Existen varios proyectos de investigación para responder a la pregunta de si es posible que a partir de un grupo de agentes tontos, tener algún tipo de inteligencia o inclusive si se puede estudiar la inteligencia a partir de estos agentes tontos?. [BAR-1]

Algunos proyectos de investigación al respecto se sustentan en el *enfoque basado en agentes tontos* [BAR-1]. Es una modificación radical del enfoque ortodoxo de la IA, que modela el comportamiento inteligente diseñando e implementando un agente complejo, en términos de programación. La efectividad de este agente complejo ha sido demostrada en dominios racionales especializados como juegos, razonamiento y planificación de trayectorias [RUS-1].

En los proyectos de agentes tontos, éstos agentes pueden estar aislados intentando optimizar su rendimiento o no estar aislados y pudiendo interactuar entre ellos. El comportamiento de los agentes puede involucrar su coordinación con otros agentes. Los dominios de los problemas incluyen el control sensorial y motriz, el estudio del

[8] Una partida de ajedrez en un torneo, se tiene como máximo 2 horas para realizar las primeras cuarenta jugadas. Dando un promedio aproximado de 3 minutos por jugada.

comportamiento social, el modelado de modos cognitivos interactivos (como el lenguaje) e, incluso, la resolución de problemas complejos integrando a dichos agentes.

Este enfoque supone la generación de una población de agentes inicialmente tontos. Dicha población suele ser grande (quizás 50, 100 o incluso 1000 agentes). Los agentes pueden ser todos idénticos o pueden existir diferencias. Suelen construirse aleatoriamente con la expectativa de que parte de la variación natural en la población permita resolver la tarea deseada. La población entera de agentes se evalúa entonces de acuerdo a su adecuación a la resolución de la tarea de interés. Los mejores agentes se suelen seleccionar para continuar su modificación. Mediante un ciclo continuo de selección y modificación surgen los agentes más adecuados para la tarea. En algunos casos, los agentes trabajan solos, cada uno de ellos intentado superar a los otros en la resolución de la tarea en cuestión. En otros casos, los agentes trabajan conjuntamente para resolverla.

Un ejemplo de este enfoque lo plantea *Gary Parker* [PAR-1], quien ha construido un modelo computacional de un robot que incluye variables para cada aspecto de las características físicas del robot. Los programas de control pueden probarse en el modelo, haciendo que se mueva un robot simulado. Para cada robot real, se calibra el modelo para que corresponda a las características del robot. De hecho, durante la vida de un único robot, la recalibración es a veces necesaria. Una vez que se ha definido el modelo del robot, se ejecuta un algoritmo genético sobre las soluciones del control motriz. Ha conseguido rutinas de control motriz "suave" que imitan la forma de andar de varios insectos diferentes [BAR-1]. Los robots tontos, incapaces de moverse de ninguna forma con sentido, evolucionan hasta robots capacitados a través de generaciones de evolución simulada. Un enfoque similar al planteado por Brooks, sobre la formación de inteligentes a partir de elementos carentes de ella.

Un GA[9] es un método de búsqueda inspirado en la selección natural [GOL-1] [BAR-1]. Una población de soluciones se genera aleatoriamente y cada solución se evalúa su adecuación, que es una medida directa relacionada con el rendimiento de la solución en el problema. Las soluciones más adecuadas se emplean para generar nuevas soluciones. La repetición de este proceso muchas veces (generaciones de soluciones que evolucionan) puede llegar a un conjunto de soluciones que resuelven bien el problema.

El problema de este enfoque es que pasa si el ambiente en el que se desenvuelve el robot cambia. La pregunta que surge entonces es cómo logramos incorporarle alguna técnica de aprendizaje para evaluar continuamente la mejor opción.

[9] Acrónimo de algoritmo genético (Genetic Algorithm)

Otro proyecto interesante es el planteado por *Christopher Baray* [BAR-2] utiliza agentes tontos con el fin de imitar los comportamientos sociales de algunos animales, para la solución de problemas. Define agentes reactivos, que respondan a estímulos, controlados por un conjunto de reglas condición - acción. Los agentes no tienen memoria, ni son capaces de modificar sus reglas durante su vida. Para permitir que la cooperación entre ellos, son capaces de emitir y recibir señales. Para decidir qué conjunto de reglas deberían utilizar los agentes, se emplean algoritmos genéticos. Poblaciones de agentes idénticos se colocan en el mundo y su esperanza de vida media representa la adecuación de las reglas condición – acción.

Aunque pueda resultar paradójico decir que comportamientos complejos pueden surgir de agentes inicialmente ingenuos, una diversidad de problemas pueden tratarse con el enfoque de los agentes tontos. Este enfoque alternativo de la IA se conoce como *softcomputing*[10], en las cuales surgen técnicas como las redes neuronales, lógica difusa, computación evolutiva, algoritmos genéticos. La gran dificultad o inconveniente de este tipo de enfoque es que se evita la representación explicita del conocimiento y el definir el por qué de la obtención de una solución particular, representa quizá una visión empírica del surgimiento de inteligencia. [KNA-1].

2.7. DISEÑO DE AGENTES.

[BES-1] propone dividir el diseño de agentes en dos etapas. La primera relacionada con la definición de los constituyentes o lineamientos del diseño entre los que se incluyen:

- Ubicación del nicho ecológico, es decir, donde se desenvolverá el agente
- Definir las tareas o conductas deseadas por parte del agente

La segunda etapa constituye el proceso en sí del diseño del agente, vinculados a la morfología, arquitectura y los mecanismos correspondientes. En esta etapa se deben resolver los siguientes puntos.

a. Agencia completa, es decir, que se cumplan con las pautas que definen a un agente autónomo.

[10] Término que pretende la distinción de los mecanismos de aprendizaje provenientes de las nuevas técnicas emergentes.

b. Paralelismo, procesos ligeramente acoplados.
c. Coordinación motriz-sensorial.
d. Diseños económicos
e. Redundancia
f. Balance ecológico
g. Valor, es decir la capacidad de subsistencia del agente en el nicho.

La relación de estas actividades se puede observar en la siguiente figura.

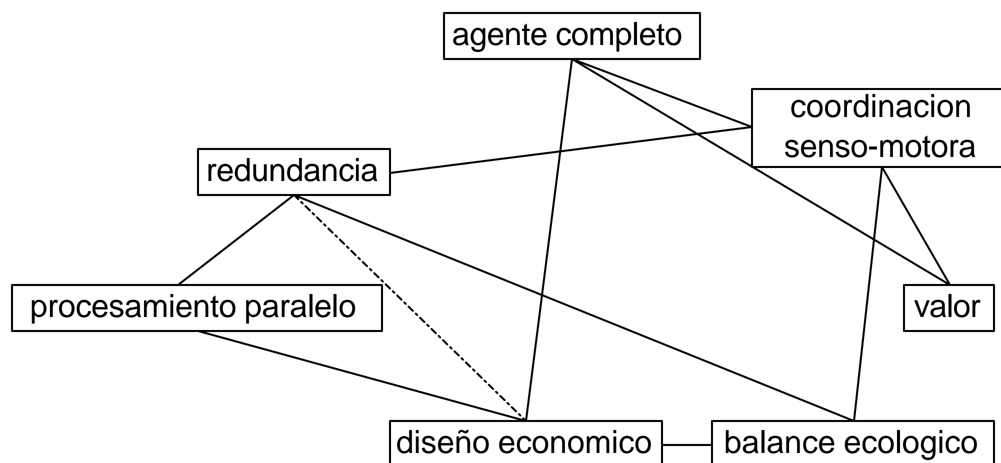

FIGURA 2.3. RELACIONES DE LOS PRINCIPIOS DE DISEÑO DE AGENTES

Un ejemplo de esta metodología se puede ver en el proyecto de la NASA, denominado ***SlutBot*** [BES-1]

Wooldridge plantea una metodología orientada al diseño de agentes de software [WOO-2][WOO-3][WOO-4][ZAM-1]

En las próximas secciones se discutirán dos lenguajes de programación, con los cuales se pueden desarrollar agentes como lo son Prolog y Java. Explorando sus características y las virtudes para la programación de agentes. Adicionalmente, se abordarán algunas de las estrategias que permitan la ejecución de programas en Prolog sobre Java y viceversa.

2.8. PROLOG

La definición de dispositivos computacionales que puedan *pensar* requiere la incorporación de elementos que permitan representar el conocimiento y su consecuente manipulación. Uno de los primeros intentos consistió en la creación de un mecanismo para implementar algoritmos basados en lógica de primer orden. De este esfuerzo nació el lenguaje de programación llamado Prolog[11], desarrollado a principios de los años 70 por Alain Colmerauer, Robert Kowalski y Phillipe Roussel.

El uso inicial de Prolog subyace en la investigación del lenguaje natural [AMZ-2], fue adoptado por la rama de la Inteligencia Artificial para el procesamiento simbólico, aunque también ha sido utilizado en aplicaciones tales como sistemas expertos, base de datos inteligentes, entre otras, así como otro tipo de aplicaciones convencionales.

Prolog es un lenguaje de programación para representar y utilizar el conocimiento que se tiene sobre un determinado dominio. Más exactamente, el dominio es un conjunto de objetos y el conocimiento se representa por un conjunto de relaciones que describen las propiedades de los objetos y sus interrelaciones.

2.8.1. IMPLEMENTACIONES DE PROLOG

Las modernas implementaciones de Prolog están basadas en una máquina virtual llamada WAM (Warren Abstract Machine) que fue implementada por David D. Warren en la Universidad de Edimburgo [AIT-1]. La WAM fue diseñada con la intención de implementar las facilidades del lenguaje lógico en una arquitectura netamente imperativa.

> *"La WAM es una máquina abstracta que consiste en una estructura de memoria y un conjunto de instrucciones que se ajustan a Prolog. Ésta puede ser implementada eficientemente sobre un amplio rango de dispositivos de hardware, y sirve como objetivo para compiladores portables de Prolog" [AIT-1]*

Hassan A¨ıt-Kaci

[11] Acrónimo de PROgramming in LOGic.

LA WAM representa el punto de partida para la creación de un compilador Prolog. La WAM utiliza una pila única donde subyacen todas las estructuras e informaciones necesarias para realizar la corrida del programa. La WAM se abordará con mayor detalle en el capítulo 3.

2.8.2. SINTAXIS DE PROLOG

El lenguaje de programación Prolog, como se mencionó con anterioridad se basa en la lógica de predicados de primer orden, pero para ser más específicos, en un subconjunto de ésta como lo es la lógica de la forma causal.

La lógica de primer de orden representa el conjunto de todas las sentencias que pueden ser construibles con la gramática que se muestra en la figura 2.4, en la que se representa dicha gramática de Prolog representada en BNF[12].

Símbolos especiales := :- | , | . | [|]

<atomo>:= <constante>(<lista_de_terminos>
<clausula>:= <atomo> :- <literales>.
<literales>:=<literal> | <literal> , <literales>
<literal>:=<atomo> | not(<atomo>)
<lista_de_terminos>:= <termino>| <termino>,<lista_de_terminos>
<termino>:=<variable>|<numero>|<constante>|
    <constante>(<lista_de_terminos>)
<variable>: <maysuculas> | <constante>-
<numero>:- 1|2|3|4|5|6|7|8|9|0
<constante>:- <minusculas> | <nurnero> | <constante> |<_>
    ∴ <minusculas> el conjunto de letras del alfabeto en minúsculas
    ∴ <minusculas> el conjunto de letras del alfabeto en mayúsculas.

FIGURA 2.4. DESCRIPCION BNF DE PROLOG.

A partir del esquema anterior podemos formar el conjunto de cláusulas que conformarían un programa en Prolog. Se puede observar que un átomo es entonces un predicado aplicado a una tupla de términos, como por ejemplo:

ganar(Y, suerte(pueblo)).

[12] Acrónimo de Backus Nahur Form

Una cláusula es entonces un átomo seguido por cero o más literales, donde un literal es un conjunto de términos.

La lógica de forma clausal, se fundamenta en la estructuración del conocimiento en forma de cláusulas, definidas en forma de predicados asociados a variables que son cuantificadas universalmente, es decir, válidas para todo el conjunto de valores posibles dentro del dominio. Un ejemplo de la forma clausal se muestra a continuación,

"X, "Y (gusta(jhon,X) or not (gusta(X, inteligencia_artificial)) &
(gusta(maria, inteligencia_artificial))&
(gusta(X,Y) or not(ama(X,Y))))

\
"　　es el símbolo de cuantificación universal para expresar su validez para todos los posibles valores de un conjunto denominado dominio de Herbrand.
OR　　es la operación lógica de o inclusiva
&　　es la operación lógica de y (and)

Podríamos traducir estas cláusulas a un conjunto de frases más cercanas a nuestro idioma cotidiano, se dice que a Jhon le gusta algo o alguien o no.

Podemos manipular las cláusulas anteriores para rescribirlas según la lógica de primer orden.

{("X) (gusta(jhon,X) or not (gusta(X, logica)) &
(gusta(ruddy, logica))&
(gusta(maria, logica))&
("X, "Y) (gusta(X,Y) or not(ama(X,Y)))) }

Se puede entonces representar el programa como un conjunto de cláusulas, reemplazando "or not" por su equivalente lógico "if", y el cual se representa en el lenguaje Prolog según la sintaxis conocida como Edimburgo, utilizando los símbolos ":-". Entonces las cláusulas se transformarían en:

```
gusta(jhon,X) :- gusta(X, logica) .
gusta(ruddy, logica).
gusta(maria, logica).
gusta(X,Y) :- ama(X,Y).
```

Estas últimas representan la forma como se escribirían las cláusulas en un programa en Prolog.

Se prefiere la lógica de forma clausal para el área computacional, en lugar de todas las capacidades de la lógica de primer orden debido a que facilita el almacenamiento en

memoria, reduce además el número de reglas de inferencia para la solución de problemas, y presenta significado computacional [HOG-1].

Podemos agregar que además que para facilitar la solución de problemas en un tiempo finito, se utiliza una regla de inferencia básica como es *modus tollens*; lo cual conlleva al establecimiento de un tipo particular de cláusula denominada de *Horn,* la cual contiene uno o menos literales negativos, como por ejemplo.

(gusta(jhon,X) or not (gusta(X, lógica))

Por lo tanto, se puede sintetizar que un programa Prolog esta compuesto por un conjunto finito de cláusulas de Horn. Prolog presenta además, dos mecanismos importantes para su funcionamiento como lo es la resolución y la unificación. En las siguientes secciones se describirá estos dos conceptos.

2.8.3. RESOLUCIÓN

Comprende una regla de inferencia aplicable a la lógica de forma clausal. La idea consiste en que dadas dos cláusulas por el proceso de resolución se puede derivar una nueva cláusula como consecuencia de éstas últimas. Para realizar la mencionada tarea se requiere de un método conocido como sustitución. La sustitución consiste en hallar los valores que permitan lograr la unificación de dos cláusulas, a través de la sustitución de variables por valores existentes en el dominio de Herbrand.

El domino de Herbrand esta formado por todos los términos instanciados con constantes (se la denominan también *términos grounded*), que pueden ser construidos usando símbolos constantes y símbolos de funciones disponibles en un alfabeto K – el cual está formado por todas las constantes existentes -, el cual representa el conjunto de todas las posibles combinaciones construibles con los símbolos constantes existentes.

El proceso de resolución inicialmente se basaba únicamente en la búsqueda dentro de la totalidad del espacio del dominio de Herbrand. El algoritmo de unificación permite podar el espacio de búsqueda a través de la implementación de un concepto conocido como Unificador Más General.

2.8.4. UNIFICACIÓN

La unificación representa una de las operaciones más poderosas de Prolog, ya que permite realizar una correspondencia o matching sobre átomos de Prolog [AMZ-2].

Existen varios algoritmos para llevar a cabo la unificación, uno de los más utilizados es el denominado Algoritmo de *Robinson*. El algoritmo parte de la suposición de que dado un par de átomos de la forma J(e1,...,en) y J(a1,...,an), donde ai y ei son términos. Si ellos son unificables entonces el resultado del algoritmo es el unificador más general, de lo contrario se produce una falla [HOG-1]. Utiliza como estructura principal una pila donde almacena pares de la forma <ei, ai>. A continuación se muestra el algoritmo en pseudocódigo.

```
// Se deben definir dos pilas S y fi
// Se supone que los términos son colocados en un pila S, con una función de la forma
// push(par(ei,ai),S), antes de empezar el desarrollo del algoritmo

final=false;
final=null;
falla=false;
While (! Final){
If (empty(S)) then
        salida = fi;
        final=true;
else
        pop(par(s1,s2),S); // S es la pila
        construir(par(e1,e2), par(s1,s2), fi) //Construye par(s1*fi, s2*f2)
        if (const(e1) != const(e2)) then final=true;
        else if (functor(e1) != functor(e2)){
                final=true;
                falla=true;
        }
        else if (functor(e1) = functor(e2)){
                push(par(términos(e1),términos(e2)),S);
        }
        else if (const(e1)& functor(e2)){
                final=true;
                falla=true;
        }
        else if (const(e2)& functor(e1)){
                final=true;
                falla=true;
        }
        else if (variable(e1) & variable(e2)){
                push(par(e2,e1),fi);
        }
        else if (variable(e1)){
                push(par(e2,e1),fi);
```

```
        }
        else if (variable(e2)){
                push(par(e2,e1),fi);
        }
        else {
                final=true;
                falla=true;
        }
}
```

Un ejemplo de la unificación puede ser visto, realizando una consulta a Prolog, por ejemplor:

```
?- padre(X,Y) = padre (edgar, jhon)
X=edgar
Y=jhon
```

2.9. JAVA

El lenguaje Java fue desarrollado por *Sun Microsystems* a partir del año 1991 como un proyecto de investigación para el control de dispositivos inteligentes, sin embargo el lenguaje no se popularizó hasta que los desarrolladores de *Sun* decidieron utilizar éste lenguaje para crear páginas Web de contenido dinámico, lo cual despertó gran interés en el creciente mercado de Internet [BOO-1].

Los programas en Java consisten en varias partes: un entorno, el lenguaje, una API[13] y varias bibliotecas de clases. La programación Java es inherentemente orientada a objetos, el código se construye sobre la base de clases y métodos, haciéndolo altamente modular y reutilizable, por lo cual existen extensas bibliotecas de clases que pueden incorporarse al código.

El código de Java se considera robusto porque no permite crear código autodestructivo, es decir, el lenguaje no posee ninguna forma directa de manejar punteros y evita que el programador sature accidentalmente los recursos del sistema. Por otro lado los mecanismos de las excepciones obligan al programador a cubrir muchas situaciones de error.

[13] Acrónimo Application Programming Interface, por su nombre en inglés

Adicionalmente se considera que Java es seguro, ya que tiene numerosos mecanismos que impiden la creación de código malintencionado, limitando la funcionalidad de los programas de acuerdo a las políticas y los privilegios de los objetos; la nueva filosofía de Java está orientada a obligar a los programadores a firmar digitalmente sus códigos de los *applets* para poder ser ejecutados y distribuidos, de manera útil en Internet, evitando la existencia de código peligroso programado en Java.

El compilador de Java crea un código llamado "*bytecode*" que es independiente de la plataforma y debe ser ejecutado por un intérprete –conocido con la máquina virtual de Java- que convierte el *bytecode* en instrucciones propias de la computadora en que se esté ejecutando. Esto permite que en teoría los programas puedan ser compilados una vez pero ejecutados en numerosas plataformas, lo cual se cumple con ciertas limitaciones.

Java es un lenguaje de programación orientado a objeto, con la característica primordial de ser portable, -gracias al formato bytecode-, ideal para la comunicación en Internet.

> *"Compílelo una vez y córralo donde sea"*
>
> Sun MicroSystems

El argumento para utilizar el Java como una herramienta de visualización de programas de control, lo sustenta el hecho de su independencia de plataforma lo que permite portabilidad de los programas, además de la facilidad de programación de interfaces gráficas.

Java soluciona el problema de los lenguajes de programación tradicionales, en cuanto a su dependencia a la arquitectura de la máquina donde se produce la compilación, a través de la definición de un código objeto que no depende de un chip particular. El resultado de una compilación de Java es el código objeto llamado bytecode. El resultado de la compilación es básicamente un flujo de bytes, consistente de códigos de operación (opcodes) y parámetros para una máquina teórica. Esta máquina es llamada máquina virtual Java (JVM[14] por sus siglas en inglés).

La JVM puede ser incorporada en cualquier presentación que se requiera. Cuando ésta es implementada en software la JVM reside entre la máquina particular y los programas que se quieran correr, como puede observarse en la figura 2.5. La JVM interpreta la fuente compilada en Java para correr una determinada aplicación.

[14] Acrónimo de Java Virtual Machine

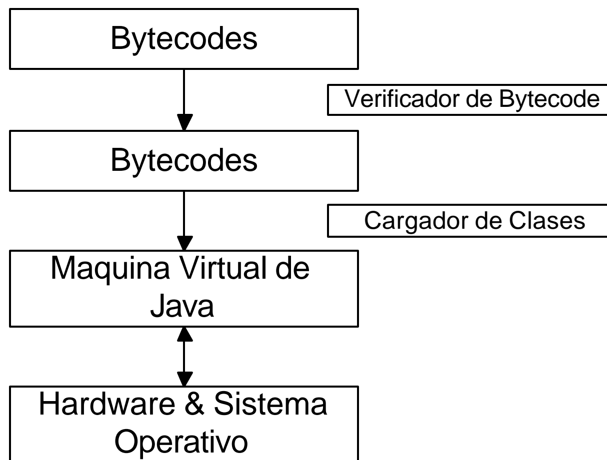

FIGURA 2.5. MÁQUINA VIRTUAL DE JAVA

El archivo resultante de la compilación en Java tiene extensión *.class*. Cualquiera otra llamada de clases dentro ese archivo creará nuevos archivos *.class*. Cuando la JVM corre, esta busca cualquier clase referenciada por la clase que se esta ejecutando y la cargará también, por lo tanto se puede decir que Java es un lenguaje dinámico, que no requiere que todas las cosas sean definidas estáticamente al momento de la compilación. En la figura 2.6 puede observarse el ciclo de desarrollo de Java [BOO-1]

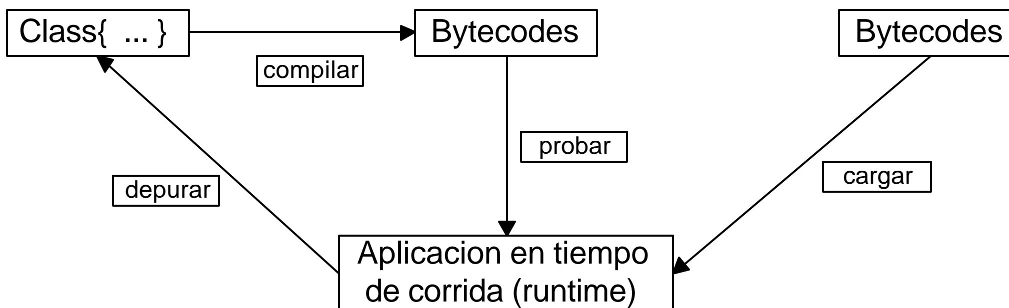

FIGURA 2.6. CICLO DE DESARROLLO DE JAVA

2.9.1. JAVA Y LA PROGRAMACIÓN ORIENTADA A OBJETOS

El principio básico de la programación orientada a objetos es que un programa se ve como una secuencia de “transformaciones” en un conjunto de objetos [BOO-1] y la interacción que ocurre entre ellos. Un objeto es una estructura formada por la combinación de datos y las operaciones que actúan sobre ellos, emulando el significado

real de objeto, ya que tienen atributos -características que lo describen- y comportamiento - conjunto de cosas que el objeto puede hacer-.

Las clases son la estructura que define el tipo de dato, en tanto los objetos son *instancias* de una clase. Dentro de un programa pueden definirse una o varias clases y crear uno o más objetos de cada clase.

La definición de la clase está formada por un conjunto *variables* y *métodos*, siendo los métodos las funciones que actúan sobre dichas variables. Las variables y los métodos pueden o no ser accedidos desde el exterior del objeto dependiendo de sus propiedades, pudiendo ser *públicos* o *privados*. Estas propiedades son establecidas en la declaración de la clase y afectan el comportamiento del objeto y controlan los mecanismos de herencia.

Una clase puede "heredar" una o más propiedades de otra clase, esto permite que una definición de clase sea parte especificación y parte implementación, haciendo posible reutilizar un código previamente elaborado e incluso compilado. Se llama *superclase* a la clase de la cual una clase hereda sus propiedades, y se llama *subclase* a la clase que hereda las propiedades de una clase dada.

En la figura 2.7 se muestra un ejemplo del patrón de herencia, así como los atributos de cada clase.

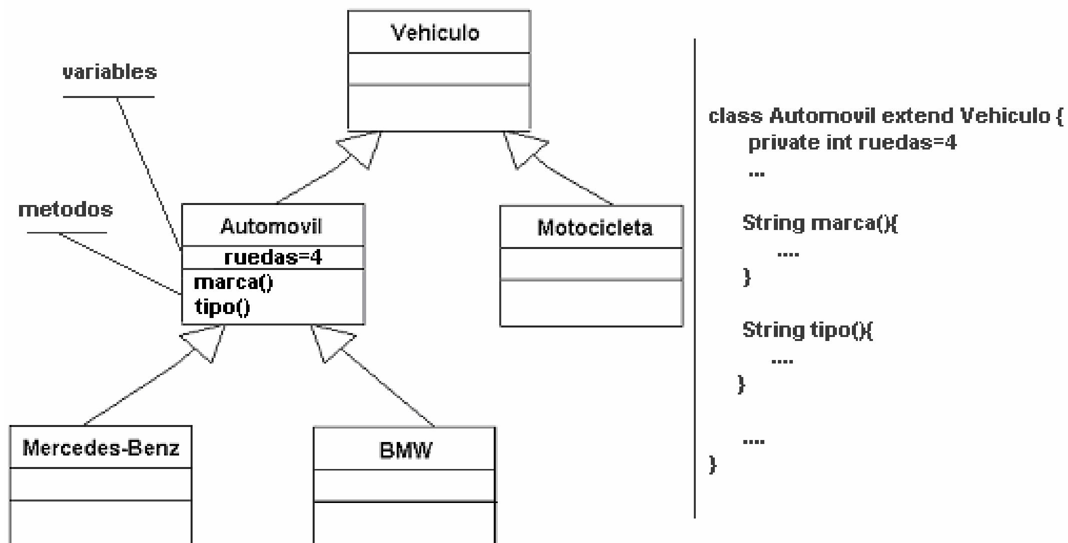

FIGURA 2.7. HERENCIA Y ATRIBUTOS DE LAS CLASES JAVA

Paquete	Descripción
java.applet	Contiene la clase *Applet* y varias interfaces para la creación de *applets*.
java.awt	Herramientas para trabajar con ventanas de Java. Creación y manipulación de interfaces gráficas de usuario.
java.io	Clases que maneja de entrada/salida de datos a un programa Java.
java.lang	El compilador incorpora este paquete automáticamente en todos los programas, contiene las clases e interfaces básicas requeridas para un programa Java.
java.net	Paquete de trabajo con redes de Java. Contiene las clases que permiten a los programas comunicarse a través de la Internet o de intrarredes corporativas.
java.util	Paquete de utilidades de Java. Contiene clases e interfaces de utilidad como manipulación de fechas y horas, capacidades de procesamiento de números aleatorios, almacenamiento y procesamiento de grandes cantidades de datos, etc.
java.swing	Esta biblioteca contiene los nuevos componentes de la interfaz de usuario de Java, permite añadir una gran funcionalidad a los programas y sustituye a la vieja biblioteca AWT, la cual se está volviendo obsoleta.

TABLA 2.B. PAQUETES PRINCIPALES DE JAVA

Toda la programación en Java está basada en objetos, la API de Java está formada por una gran colección de clases y métodos que constituyen esqueleto de todas las aplicaciones. La sentencia *import* se utiliza para incluir las diferentes bibliotecas en el código del programa, estas bibliotecas se dividen en directorios y subdirectorios de acuerdo a la herencia existente entre las clases, algunas de las bibliotecas más importantes de la API de Java [NAU-1] se muestran en la tabla 2.b.

En Java la herencia se realiza a través de la sentencia *implements* la cual permite incluir en una clase ciertas características de otra clase. Java no soporta herencia múltiple, es decir una clase sólo puede heredar propiedades de una *superclase* a la vez, sin embargo puede *implementar* varias interfaces al mismo tiempo. En Java una interfaz es una colección de definiciones de métodos que pueden ser implementadas por cualquier clase en cualquier lugar [NAU-1].

Una aplicación en Java está formada por una clase que posee una función especial llamada *main*. Cuando el programa se inicia el intérprete busca primero el método *main* y lo ejecuta.

Existen numerosos paquetes para el desarrollo de programas de Java, Sun Microsystems distribuye de manera gratuita el paquete JDK[15] que incluye una serie de herramientas[16], en la que destaca el programa *javac* que permite compilar todo el código fuente Java y generar el correspondiente *bytecode*. Para que la compilación sea exitosa el archivo con el código fuente debe llamarse de forma idéntica a la clase principal más la extensión *.java*.

2.9.2. RECURSOS

Para programar en Java es necesario contar con un compilador y las bibliotecas de la API de Java, además de un intérprete que ejecute los programas, como se mencionó anteriormente el entorno de programación es proporcionado por la empresa *Sun Microsystems* a través del JDK. A partir de la versión 1.3 de Java el entorno de programación es llamado J2SE[17]. Las últimas versiones de JDK o J2SE de *Sun* además de la documentación pueden conseguirse en [JAV-2].

Microsoft dispone también de una versión del JDK llamada SDK[18], cuya versión 4.0 ya está disponible en [JAV-3]. Esta versión incluye además bibliotecas especiales propias de Microsoft como las empleadas por los mecanismos de seguridad para *Internet Explorer*.

Existen ambientes de desarrollo que pueden ser utilizados para la programación en que presentan interfaces graficas, despliegue del árbol de clases, depuradores entre otras herramientas; que facilitan la creación de aplicaciones y *applets* en Java. Entre los ambientes existentes se encuentran: Kawa[19], VisualAge, VisualCafe[20], entre otros.

---

[15] Acrónimo de Java Development Kit

[16] Entre otros están el jar, java, javac, javadoc, *appletviewer*, etc.

[17] Acrónimo de Java 2 Standard Edition, conocido comercialmente como Java 2

[18] Acrónimo de Software Development Kit

[19] Puede ser visitado en la página web http://www.tek-tools.com/kawa

[20] Puede ser visitado en la página web http://www.VisualCafe.com

2.9.3. LA INTERFAZ DE USUARIO

La GUI confiere al programa un *aspecto* y una *sensación* distintivos, permitiendo la interacción entre el usuario y el programa. Está constituida a partir de *componentes*, los cuales son objetos visuales con los que el usuario puede interactuar a través del ratón o del teclado [BOO-1][NAU-1]. Tradicionalmente las clases que se usan para crear la GUI se encuentran en la biblioteca AWT[21], sin embargo estas clases están siendo desplazadas por los componentes de la biblioteca *javax.swing* que forman parte integral de la API de Java2.

La mayoría de los *applets* tienen una GUI, esto es una consecuencia natural de que los *applets* aparecen en la ventana de un navegador. Ya que la clase *Applet* es una subclase de la clase *Panel* de AWT, crear la interfaz gráfica de usuario es incluso más sencilla que hacerlo en una aplicación debido a que la ventana del *applet* –la ventana del navegador - ya está creada [BOO-1][JAV-4].

Adicionalmente a la interfaz gráfica de usuario los *applets* pueden crear otros tipos de interfaces de usuario, dependiendo de la información que necesiten dar o recibir, por ejemplo algunos *applets* reproducen sonidos, tanto para dar al usuario una respuesta como para crear ambientación. Otra forma de interacción con el usuario es a través de los parámetros que define, con los que se pueden obtener ciertas opciones de configuración; para dar información tipo texto al usuario un *applet* puede usar su GUI o mostrar un corto mensaje de estado en la salida de estándar, en algunos navegadores estos mensajes se ven en la barra de estado.

Gran parte de la funcionalidad de los componentes derivan de la clase *Component* o de la clase *Container*, toda clase que hereda de la clase *Component* es un *componente*. La jerarquía de herencia de estas clases determina la forma en que se comportan cada uno de los objetos que conforman un *applet*. En la figura 2.8 se ilustra la jerarquía de herencia de los diferentes componentes que se relacionan con un *applet* [BOO-1].

[21] Se encuentra en el paquete *java.awt*

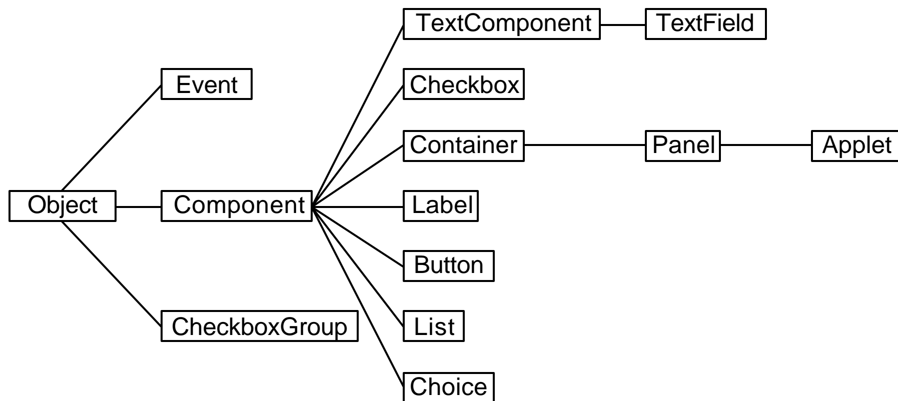

FIGURA 2.8. JERARQUÍA DE HERENCIA DE LOS COMPONENTES DE JAVA.

2.9.4. EVENTOS

Los eventos son acciones asíncronas que ocurren durante la ejecución del programa. Un usuario puede generar eventos cuanto interactúa con la GUI, por ejemplo cuando pulsa un botón o ingresa datos en un cuadro de texto. Estos eventos son manejados por Java, como objetos que se crean en el momento de la generación del evento y que poseen información acerca de la naturaleza y detalles del mismo.

En el siguiente ejemplo se ilustra un programa sencillo que crea una interfaz de usuario muestra los eventos asociados. La interfaz del programa se muestra en la figura 2.9.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class suma extends Applet implements ActionListener{
  Label lbl_num1,lbl_num2,lbl_suma;
  TextField txf_num1,txf_num2,txf_suma;
  Button btn_suma;

  public void init()  {
   lbl_num1 = new Label("x   = ");
   lbl_num2 = new Label("y   = ");
   lbl_suma = new Label("x+y = ");
   txf_num1 = new TextField("0",8);
   txf_num2 = new TextField("0",8);
   txf_suma = new TextField("0",8);
   btn_suma = new Button("Sumar");

   add(lbl_num1);
   add(txf_num1);
```

```
  add(lbl_num2);
  add(txf_num2);
  add(lbl_suma);
  add(txf_suma);
  add(btn_suma);
  btn_suma.addActionListener(this);
 }

public void actionPerformed(ActionEvent event)
{
  int x=0,y=0,suma;

  try   {
    x = Integer.parseInt(txf_num1.getText());
    y = Integer.parseInt(txf_num2.getText());
  }  catch(NumberFormatException e) {
    x = 0;
    y = 0;
  }   finally    {
    suma = x+y;
    txf_suma.setText("" + suma);
  }
 }
}
```

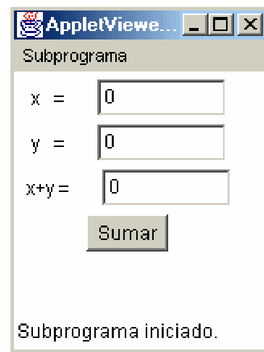

FIGURA 2.9. INTERFAZ DE USUARIO DEL APPLET SUMA.

Un componente puede tener varios manejadores de eventos, de esta manera una misma acción puede generar diferentes respuestas al mismo tiempo; además de esto, un mismo manejador de eventos puede ser agregado a diferente componentes, de manera que componentes diferentes generen la misma respuesta. De esta forma se crean interfaces que interactúan con el usuario de forma compleja con una lógica de programación clara y flexible.

2.9.5. HEBRAS O HILOS

Una hebra[22] o hilo, algunas veces llamado *contexto de ejecución* o *proceso liviano* es un flujo secuencial simple de control dentro de un programa. Las hebras pueden ser usadas para aislar tareas, cuando se corren uno o más tipos de *applets* estos crean una hebra para realizar cada tipo de operación. Cada hebra es un flujo secuencial de control dentro de una aplicación, donde las operaciones de cada hebra corren independientemente y al mismo tiempo.

Una hebra funciona de manera similar a un programa, tiene un principio, una secuencia y un fin en cualquier momento dado durante su tiempo de vida, sin embargo una hebra no es un programa, puesto que no puede correr por sí misma, las hebras corren dentro de un programa.

La gran ventaja de las hebras es que varias de ellas pueden ejecutarse simultáneamente dentro del mismo programa, esto simplifica enormemente la lógica de programación, sobretodo en el manejo de procesos asíncronos.

Existen varias maneras de crear hebras en Java, una de ellas es crear una clase que implemente la interfaz *Runnable* y un objeto de la clase *Thread* que la ejecute [NAU-1].

2.9.6. EXCEPCIONES

Una excepción es un evento que ocurre durante la ejecución del programa que rompe con el flujo normal de las instrucciones [JAV-4]. Muchos tipos de errores causan excepciones, desde problemas por serios errores de hardware como el bloqueo repentino del disco duro hasta errores simples de programación, así como tratar de acceder a elementos fuera del rango de un vector, por ejemplo. Cuando tales errores ocurren dentro de un método de Java, el método crea un objeto de tipo *excepción* y lo maneja fuera del sistema de ejecución. Un objeto *excepción* contiene información acerca del error ocurrido, incluyendo su tipo y el estado del programa en el momento de ocurrir. El sistema de ejecución responsable de encontrar algún código para manejar el error, en la terminología

[22] En inglés Thread

de Java crear un objeto excepción y manejarlo dentro del sistema de ejecución se conoce como *lanzar una excepción*.

Después de que un método lanza una excepción, el sistema de ejecución entra en la búsqueda de alguien que maneje la excepción. El juego de posibles *quienes* que manejen la excepción es el conjunto de métodos en la pila de llamadas, empezando con el método en el que ocurrió el error. Un manejador de excepciones es considerado apropiado si maneja el mismo tipo de excepción que fue lanzado, de esta forma la excepción *sube* a través de la cola de llamadas hasta que encuentra un manejador apropiado para la excepción. Cuando se encuentra un manejador de excepción adecuado se dice que el manejador *captura la excepción*.

Si dentro de un programa se emplea un método que puede lanzar una excepción, debe existir un bloque *try – catch* para manejar la excepción en caso de que aparezca, de otra forma el programa generará un error en tiempo de ejecución. Por lo tanto, Java obliga a los programadores a manejar todas las posibles situaciones de error inherentes en los métodos de la API. A continuación se presenta un ejemplo para ilustrar el manejo de las excepciones:

```
public void UnMetodo(String nombreArchivo){

  try   {
     LeerArchivo(nombreArchivo);
     . . .
   }  catch(IOException e)  {
    // Aquí se manejan los errores de lectura / escritura ocurridos
    // en el método LeerArchivo
    . . .
  }  finally   {
    // Este código se ejecuta ocurra o no una excepción
    . . .
  }
}
```

2.9.7. VERSIONES DE JAVA

Cada versión de Java incluye nuevas mejoras y nuevas herramientas, la API es una biblioteca extensa que es revisada constantemente, eventualmente incluyendo nuevos métodos y mejorando los existentes, algunos métodos son considerados obsoletos y son *desaprobados*, aún cuando siguen formando parte de la API por razones de compatibilidad hacia atrás.

En diciembre de 1998 *Sun Microsystems* dio a conocer el nombre "Java2" con la salida de su primer producto de ésta tecnología JDK 1.2, el cual fue llamado luego J2SE 1.2. La versión actual es Java2 versión 1.4.x, la cual es ya bastante diferente a la vieja JDK 1.1. Los aspectos más relevantes se encuentran los relativos a la nueva API para la interfaz gráfica de usuario y la nueva arquitectura de seguridad.

2.10. ¿JAVA Y PROLOG, UNA ALTERNATIVA?

Existe una fuerte motivación por combinar lo mejor de los mundos de la programación lógica, a través del lenguaje Prolog y la programación orientada a objetos, en especial del lenguaje Java [CAL-1][SIC-1]. **Calejo** en el documento presentado en la *PACLP 99* denominado "Java+Prolog: A Land of Opportunities" desglosa de manera coherente los puntos principales según los cuales se pueden *"integrar"* estos dos lenguajes.

Prolog no ha sido un lenguaje altamente difundido en los ambientes comerciales, tal vez debido a su carácter generalmente académico, pero continua evolucionando rápidamente, y es usado por los investigadores y desarrolladores buscando explotar las ventajas que presenta la programación lógica.

Pero hoy día la programación orientada a objeto tiene una mayor cantidad de adeptos, debido a la facilidad que brinda para definición de interfaces graficas, reutilización de código entre otras, adicionalmente del potencial para el manejo de conexiones y creación de aplicaciones de red. Uno de estos lenguajes es Java. Java permitiría fortalecer y complementar la programación lógica, como lo ha demostrado la existencia en el mercado de interfaces que integran los lenguajes de Java y Prolog.

El Prolog es un lenguaje declarativo / imperativo, que posee como mecanismos básicos la unificación y el *backtracking*. Existe una versión de Prolog que incorpora la solución de restricción denominado CLP. La versión estándar de Prolog fue propuesto por ISO[23] [ISO-1].

La plataforma Java [JAV-2], es un sistema de programación orientada a objeto en la cual esta basado en programación imperativa y el envío de mensajes, y es independiente de

[23] Acrónimo de Organización de estándares internacionales (en inglés International Standards Organization)

un sistema operativo particular. Presenta una colección grande de APIs entre GUI, gráficas 3D y comunicaciones de Internet.

La mayoría de los proyectos que combina Java y Prolog, explotan las características de reflexión y serialización que posee el lenguaje Java. Por definición, la reflexión permite el envío de mensajes de un lenguaje a otro [CAL-2] sin necesidad de programación adicional. La serialización[24] es un mecanismo abierto y uniforme para el intercambio de datos entre ambos lenguajes, pudiéndose enviar el estado de un programa a través de un flujo de bytes tipo serial.

Dadas las propiedades de Prolog y Java, se combinar para aprovechar lo mejor de ambos mundos?. Se tienen algunas alternativas para responder a la pregunta anterior.

a. Java en Prolog
b. Prolog en Java
c. Java y Prolog en un ambiente

2.10.1. JAVA EN PROLOG

La programación orientada a objetos junto a la programación lógica, permite la disposición de mayor poder que en cualquier sistema Prolog convencional, ya que tiene una capa extra de elementos bajo orientación a objetos, disponible para la representación del conocimiento. Pero, la realización de interfaces con el mundo todavía no resultan económicas, en el sentido del esfuerzo y el tiempo invertido para su concreción. Un descripción de un proyecto en el cual se agrega la capacidad de la orientación a objetos a Prolog puede ser visto en [OOL-1].

2.10.2. PROLOG EN JAVA

Existen varios proyectos que implementan Prolog en Java como los son: DGKS Prolog [DGK-1], JavaLog [JAV-1], Jinni [JIN-1], MINERVA [MIN-1], Prolog Café [PRO-1], W-Prolog [WPR-1], jProlog [JPR-1] entre otros.

[24] La serialización también se conoce como Persistencia de Objetos.

Realizar un sistema Prolog en Java tiene las siguientes ventajas: portabilidad, movilidad de código y flexibilidad. Sin embargo, la principal dificultad de esta aproximación es que la máquina virtual de Java es diferente a la máquina abstracta de Warren, y por lo tanto la eficiencia de implementación se ve disminuida. Aunque existen alternativas para mejor el performance de una implementación de Java en Prolog [TAR-1]. Un reporte sobre estas diferencias son discutidas en [OFI-1].

Existen algunos aspectos que deben ser abordados a la ahora de implementar alguna forma de Prolog en Java, entre los que destaca:

a. ¿Cómo podemos controlar archivos .class?, ya que la máquina virtual de Java no brinda acceso al contador de programa del microprocesador, como lo hace la WAM.

b. El recolector de basura[25] de Prolog es más eficiente que el de Java

c. Los términos de Prolog como objetos "pierden" las optimizaciones de tiempo y espacio hechas por la WAM, además JVM esta optimizada con respecto a C++, no para Prolog.

d. La sobrecarga en el tiempo de corrida, es decir, existe un chequeo extra en cuanto a los límites del arreglo de trabajo de la WAM.

Los anteriores puntos han llevado a que los actuales proyectos sean lentos en referencia a un sistema Prolog convencional. Adicionalmente son objeto de estudio las posibles formas de atacar estas restric ciones [CAL-1]

Existen varios proyectos vinculados con el manejo de Prolog en Java, como se mencionó anteriormente, pero vamos a describir dos de los proyectos más representativos, debido al soporte al usuario y la continua investigación; como lo son: Minerva y Jinni.

2.10.2.1. MINERVA

El sistema Minerva, fue desarrollado por *IF Computer*, posee una implementación total del Prolog propuesto por ISO en Java, más la inclusión de algunos paquetes adicionales,

[25] Garbage Collector por su nombre en inglés

como predicados declarativos para las GUI, entre otros. Posee compatibilidad con los *applets*, adicionalmente se pueden ejecutar múltiples máquinas Minervas que se compilan en un ByteCode propio – Minerva ByteCode –.

Por ejemplo, minerva puede agregar un constructor de dos formas:

a. Las declaraciones de forma estática
b. :- get_class(´java.lang.String', String),
   get_class(int Integer),
   get_method(String, substring, [Integer,Integer], Substring),
   invoke_method(Substring, hello, [1,4], Result).

Este último ejemplo permite imprimir la palabra “hola”.

Las principales clases definidas por Minerva son:

a. Minerva, que representa la máquina virtual.
b. MinervaTerm, MinervaAtom, MinervaLong, entre otras. Definen los constructores básicos de la máquina virtual.
c. MinervaObject define un objeto para la máquina virtual Minerva.

Un ejemplo de cómo podría llamarse Minerva desde Java seria:

```
// Declaración de una máquina virtual Minerva
Minerva engine = new Minerva(applet, args)
//
…
…

…
// Ejecución de una  instrucción
if(engine.execute("append", L1, L2,V))
…
//
```

En el ejemplo anterior podemos observar como se instancia una clase Minerva, que se comporta como la máquina virtual para interpretar el código correspondiente. Luego de inicializar la máquina Minerva, se implementan las instrucciones lógicas utilizando las clases de la biblioteca Minerva.

2.10.2.2. JINNI

El programa Jinni, asume la programación lógica como una especie de "*internet glue*" para aplicaciones con bases de conocimiento distribuidas. Es una máquina Prolog liviana y compatible con los *applets*. Es mucho mejor que un sistema Prolog convencional, ya que posee múltiples máquinas, que se reflejan en una hebra de Java. Utiliza la técnica de "*pizarrones*" para sincronización en Java con un sistema llamado *Linda*. Adicionalmente, presenta llamadas remotas de alto nivel a Jinni o BinProlog [BIN-1]

Jinni presenta buena integración de los mecanismos de Java en una implementación distribuida de Prolog. En la figura siguiente se muestra el front-end de un *applet* tipo Jinni.

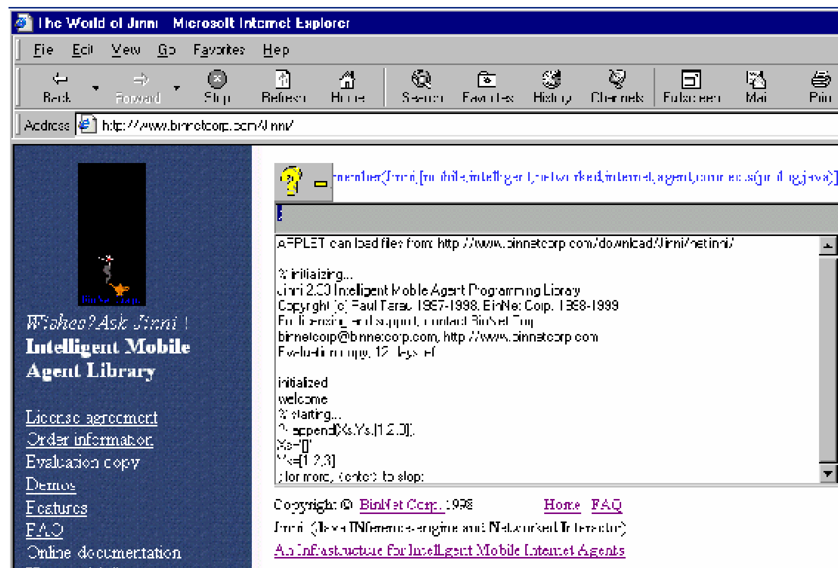

FIGURA 2.10. FRONT-END DEL APPLET JINNI

A continuación un ejemplo de la construcción de una aplicación Jinni

```
Class sumvec2 extended UserRunBuilder{
        sumvec2() {super(s);}

        Public int chec(Areg p){
                Run v1= (Run)getArg(2);
                Run v2= (Run)getArg(1);
                Double v1x=((Num) v1.getArg(2)).getValue();
                Double v1y=((Num) v1.getArg(1)).getValue();
                Double v2x=((Num) v2.getArg(2)).getValue();
                Double v2y=((Num) v2.getArg(1)).getValue();
                Run r = new Run("v", new Real(rx), Real(ry));
                Return bothArg(2,r,p);
        }
```

```
}

class UserBuilding{
        static void addBuilding(){
                register(new sumvec2);
        }
        ...
}
```

El programa anterior, describe la suma de dos vectores utilizando las bibliotecas de Jinni, y que permite la creación posterior del front-end como un applet de Jinni. Se pueden llamar las bibliotecas de *Prog* y *ProgBlackBoard* para interpretar el código Prolog. En [JIN-1] se desaglosa con mayor especificidad los mecanismos para la generación de código Jinni.

2.10.3. PROLOG MÁS JAVA

Escribiendo aplicaciones en Prolog y Java, tiene la ventaja de captar lo mejor de ambos paradigmas, es decir, unificación, backtracking, restricciones, adicional al envío de mensajes y las posibilidades de correr en múltiples plataformas. Cosecha los beneficios de décadas de investigación en el área de la programación lógica, y se aprovecha de la inversión realizada por Java.

Pero, existen algunos asuntos que deben ser tratados a la hora de implementar lo mejor de ambos mundos, como por ejemplo:

a. ¿Debe pertenecer todo a un mismo proceso del sistema operativo?.
b. ¿Qué pasa con la granularidad de las comunicaciones?
c. ¿Quién llama a quien, Prolog o Java? ¿Existen algunas restricciones?
d. ¿Qué sucede con la conversión de datos?. ¿Cuáles con las relaciones entre los términos / relaciones y los objetos?.
e. ¿Cuál seria el impacto sobre los sistemas existentes?.

Los programas que implementan Java y Prolog, se pueden clasificar en dos grandes grupos: los basados en JNI y los basados en sockets, según Calejo [CAL-1]. En la tabla siguiente se muestra algunos de estos programas:

Basados en JNI	Basados en sockets
Amzil [AMZ-1]	InterProlog de XSB [INT-1]
Jasper de SicsTUS [JAS-1]	Prolog IV
JPL de SWI	
JIPL de K-Prolog [JIP-1]	

TABLA 2.C. PROGRAMAS JAVA Y PROLOG

En las siguientes secciones se describe el programa más representativo de cada una de tendencias para la implementación de Prolog y Java, así como los mecanismos básicos para la generación de los códigos correspondientes.

2.10.3.1. JASPER.

Esta basado en un interfaz JNI[26] [JNI-1], que es un programa adicional que se inserta en Sicstus [SIC-1] como una interfaz importada. Un emulador Sicstus se invoca por la JVM, que permite la ocurrencia de invocaciones simétricas: JVM carga el emulador y viceversa.

A continuación un ejemplo de la invocación de Java y Prolog desde Jasper. Para llamar a un programa en Java desde Jasper se realizaría de la siguiente forma:

```
public class Simple{
        static int simpleMethod(int value){
                        return value*42;
        }
}

:- load_foreign_resource(simple).

foreign(method('Sirnple´, 'simpleMethod', [static]), java, simple(+integer,[-integer])).

foreign_resource(simple, [ method('Simple', 'simpleMethod', [static]).
```

[26] Acrónimo de Java Native Interface

Para llamar a un programa Prolog desde Jasper se realiza de la siguiente forma:

```
//se omite la declaración de variables
sp =new SICStus(argv, null);
sp.load("train.ql");
pred =new SPpredicate(sp,"connected", 4,"");
to = new SPTerm(sp, "Merida");
from = new SPTerm(sp, "Sancristobal");
way = new SPTerm(sp).putVariable();
query = sp.openQuery(pred, new SPTerm[] { from, to, way, way });
while (query.nextSolution()) {
        System.out.println(way.toString());
}
```

Este programa resuelve el problema del viajero en Prolog, determinando cual es la mejor ruta entre Sancristobal y Merida. Se utiliza el archivo fuente de nombre *ciudades.ql*, donde se encuentra la base de datos de las ciudades de Venezuela, como punto de partida. En el programa Jasper, se define la meta a través del método *openQuery,* y se realiza las iteraciones correspondientes, mientras el método *nextSolution* retorna una respuesta (es decir, el mejor camino de Sancristobal a Merida).

2.10.3.2. INTERPROLOG

El otro programa es InterProlog de XSB Prolog, que se funciona enlazando un programa Prolog en Java utilizando los puertos de comunicación estándar de TCP/IP[27], es decir, los sockets. El programa Java lanza máquinas Prolog correspondientes de forma similar como lo hace el sistema Minerva.

Hace uso intensivo de la reflexión y serialización de Java, para la comunicación de Java con Prolog. Es fácilmente portable pero esta diseñado específicamente para XSB Prolog.

En la figura 2.11, se muestra la información acerca de la arquitectura XSB/Java

[27] Acrónimo de Transport Control Protocol / Internet Protocol

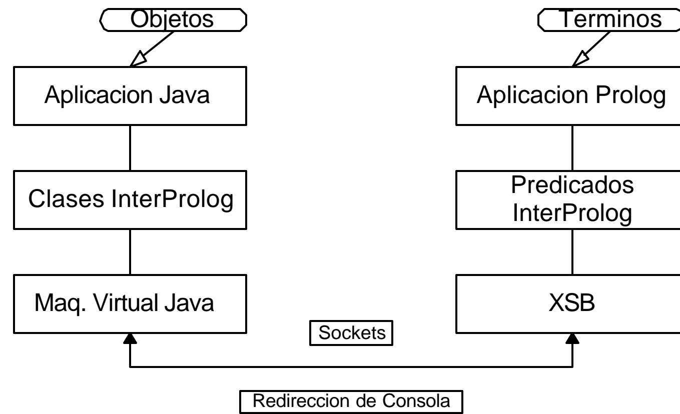

FIGURA 2.11. ARQUITECTURA XSB / JAVA.

Ahora bien, como puede observarse en la figura anterior, el programador está en la capacidad de realizar programas tanto en Prolog como Java y ejecutarlo como código nativo del lenguaje correspondiente o en su contraparte. En las siguientes dos secciones se bosqueja los procedimientos generales para la ejecución de programas Prolog o Java, y su posterior ejecución en el lenguaje yuxtapuesto.

2.10.3.2.1. *PROGRAMANDO DEL LADO DE JAVA.*

Como se mencionó anteriormente se utiliza máquina Prolog, de forma que una instancia de la máquina Prolog involucra un subproceso de XSB. Presenta además, las siguientes instrucciones básicas:

IsAvailable()
sendAndFlush(String s)
deterministicGoal(String G, String RVars, Object[] bindings)
registerJavaObject(Object x)
interrupt(), shutdown()

Un ejemplo de un programa de Java con Prolog, se muestra en la figura 2.12.

```
public class TopLevelPanel extends Panel implements Prolog OutputListener{
	TextArea prologOutput;
	TextField prologInput;
	trasient PrologEngine engine;

	public TopLevelPanel(){
		System runFinalizerOnExit(true);
		setLayout(new BorderLayout());
		prologOutput = new TextArea(20,40);
		prologInput = new TextField(40);
		add("Center", prologOutput);
		add("South", prologInput);
		engine = new PrologEngine();
		engine.addPrologOutputListener(this);
		prologInputaddKeyListener(new KeyAdapter(){
			public void keyPressed(KeyEvent e){
				if(e.getKeyCode() == KeyEvent.VK_ENTER){
					e.consume();
					prologOutput.append(prologInput.getText() + "\n");
					engine.sendAndFlush(prologInput.getText() + "\n");
					focusInput();
				}
			}
		};
		focusInput();
	}

	//PrologOutputListener methods
	public void promptWasOutput(){
		// Don't care about XSB prompts
	}

	public synchronized void print(String s){
		prologOutputappend(s);
	}

	void focusInput(){
		prologInput.SelectAll();
		prologInput.requestFocus();
	}

}
```

FIGURA 2.12. PROGRAMA JAVA EN INTERPROLOG

La ventana de salida del programa anterior se puede observar en la figura 2.13

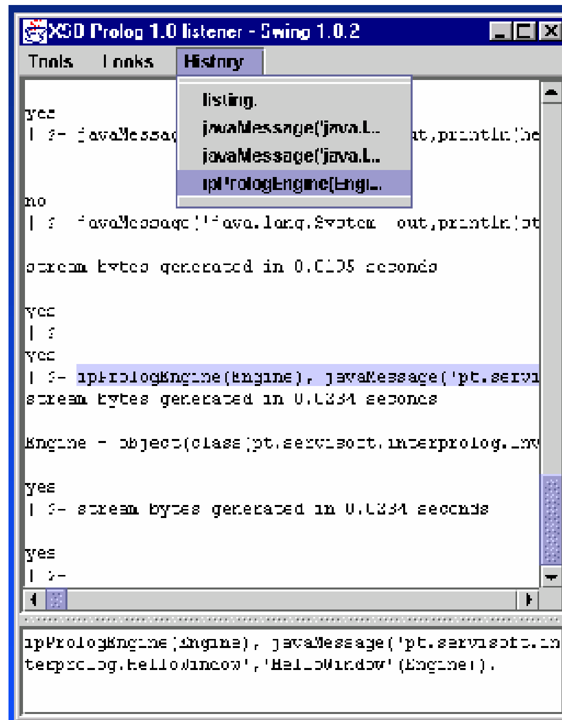

FIGURA 2.13. SALIDA DEL PROGRAMA INTERPROLOG

La interfaz entre el objeto y el término se lleva a cabo utilizando DCG[28], es decir, una gramática de cláusulas definidas. De esta forma Prolog puede reconoce los objetos Prolog en su formato serializado, además de estar en capacidad de generarlos.

Los términos en DCG son bytes u *object stream*. La salida semántica de DCG, o lo que es lo mismo, el término DCG; asume el rol de una representación o especificación para objetos en el lado de Prolog.

A continuación un ejemplo de una especificación de un objeto.

In Java : new Integer(A) // wrapper for mt E object(
class(java.lang.Integer.long(18,226,160,164,247,129,135,56),
classDescInfo([int(value)],2,class(java.lang.Number,
long(134,172,149,29,11,148,224,139), classDescInfo([],2,null)))), []+ [] + [A]
)

[28] Acrónimo en ingles de Definite Clause Grammar

Pero la pregunta es ¿como construir las especificaciones de un objeto?. Para ello InterProlog implementa dos aproximaciones:

1. El programador Java da un objeto par, definiendo solamente las variables representativas.

2. En un arranque, el programador Prolog obtiene un predicado para especificar objetos similares.

A continuación un ejemplo de máquina Prolog de InterProlog.

```
engine.deteministicGoal("ip~append([97],[99],L),countList(L,N),
ipObjectSpec('java.lang.Integer',Integer,[N],_),name(A,L)", "[1nteger,string(A)]", bindings);

// variables
//...bindings[0]==2, bindings[1]==ac
```

2.10.3.2.2. *PROGRAMANDO DEL LADO PROLOG.*

Se programa a través de un proceso de cónsola, utilizando las especificaciones de objetos tipo *helpers* (ayudantes) y con el manejo de los eventos en Java para la definición de las metas principales. La API es ampliamente abierta, tanto como la seguridad de Java lo permite.

Los mensajes son manejados utilizando la primitiva *javaMessage*, de la siguiente forma:

javaMessage(Target, Message(Args))

por ejemplo, javaMessage('java.lang.System.out',println(string('hello, world!'))) genera una salida de la forma:

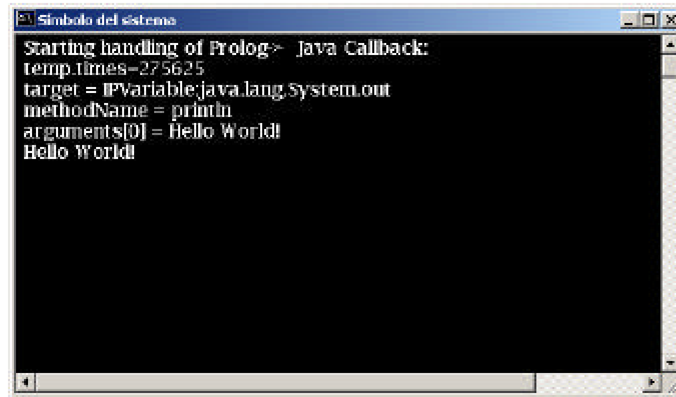

FIGURA 2.14. SALIDA DEL PROGRAMA PROLOG EN INTERPROLOG

Los predicados manejados por *javaMessage* como una hebra en Java, posee los siguientes puntos importantes:

1. *Destino*: Una clase, una variable de clase, referencia a un objeto o una nueva especificación de un objeto.
2. *Mensaje*: Puede ser cualquier método público de una clase destino.
3. *Argumentos*. Referencia a objetos existentes, nuevas especificaciones de un objeto, o un tipo básico de envoltura.

Una descripción de las potencialidades de InterProlog puede ser vista en [CAL-2]

CAPÍTULO 3. DESARROLLO CONCEPTUAL

3. DESARROLLO CONCEPTUAL

En el presente capítulo se abordará la descripción de los diferentes métodos, así como la justificación de las decisiones de diseño tomadas para el desarrollo del software de generación de clases Java a partir de programas lógicos. Adicionalmente se explicarán los mecanismos de control propios de los lenguajes declarativos que han sido incorporados en las clases Java. En primer lugar se atacarán los aspectos relacionados con la máquina abstracta de Warren. Posteriormente se describirá la estructura de la clase Java asociada a predicados, junto con los detalles del modelo de control de las clases así como la definición de las clases funcionales. Por ultimo, se abordará la descripción de la arquitectura del agente así como los diferentes programas diseñados específicamente para su funcionamiento.

3.1. WAM Y LOS MECANISMOS ESENCIALES DE DISEÑO DEL COMPILADOR PROLOG.

Como se mencionó en el capítulo 2, la Máquina Abstracta de Warren suele ser la referencia basica al implementar los compiladores de Prolog. Consiste básicamente en la manipulación de los términos en Prolog, que se convierten en registros, los cuales son agrupados en una secuencia de elementos almacenados en una estructura en forma de pila.

> *"…WAM especifica un conjunto de operaciones posibles sobre… [la] estructura de datos [conocida como registros]. Con WAM se relacionan las operaciones de manipulación simbólica que requiere Prolog con las operaciones que permite el hardware y así, es posible optimizar la ejecución del código para ese hardware…".*

Jacinto Dávila

A pesar de que la idea fundamental del funcionamiento de Prolog reside en una búsqueda secuencial sobre la estructura conocida como *head*, se han realizado experimentos para tratar de paralelizar el modo de búsqueda de este lenguaje declarativo. Un ejemplo de una implementación paralela de Prolog la podemos ver en [PLO-1], en el cual, el mecanismo de resolución se lleva a cabo de forma paralela. La información sobre la

transformación de términos es una estructura de pila, basado en C y la cual se distribuye en diferentes nodos.

Ahora bien, bajo un esquema optimizado para la arquitectura de una determinada máquina como en el caso de WAM, resulta complejo pensar en una arquitectura abierta y distribuida que implemente los mecanismos como resolución y unificación propios de los lenguajes declarativos. Una descripción en forma de objetos podría resultar una alternativa viable para una futura implementación distribuida de los mecanismos de los lenguajes declarativos [CAL-2].

Para el diseño de un compilador utilizando como base una WAM, se requiere la definición de los mecanismos conocidos como directivas. Las directivas son procedimientos que permite almacenar los términos Prolog, definición de los registros, ejecución de la unificación, manejo de las variables libres, entre otras varias tareas. Un compendio de las directivas que deben definirse para la implementación de la WAM se discute en [AIK-1].

En [AIK-1] se explica además, en forma detallada el método seguido para la implementación de una Máquina Abstracta de Warren. La idea central de esta metodología consiste en definir progresivamente diferentes etapas de diseño para la consecución de una WAM funcional y efectiva. La primera etapa de diseño consiste en la definición de las funciones básicas, entre las que podemos mencionar almacenamiento y búsqueda de términos, la unificación y la resolución. Adicionalmente se lleva a cabo una diferenciación entre el programa almacenado en memoria y la forma de manipular una consulta a un programa dado.

La manipulación de los términos su almacenamiento y búsqueda, se orienta a la sintaxis estándar de Prolog [ISO-1], que se expresa en la figura 2.4 utilizando la forma BNF.

En cada una de las etapas siguientes se lleva a cabo una mejora significativa de los procedimientos descritos en la primera etapa, así como la inclusión de nuevas capacidades al compilador. La segunda etapa consiste en la definición de nuevos mecanismos que son incorporados a la máquina desarrollada en la primera etapa, que permita leer un archivo donde reside o permanece el programa en Prolog, y almacenar las diferentes cláusulas en memoria. Esta operación es evidente al momento de realizar una operación de *consult/0* en un IDE de Prolog típico. Adicionalmente, en esta etapa se “retocan” los procesos de unificación y resolución, así como, el procedimiento para la definición de los mecanismos de traducción de la consulta o *query*.

En una tercera etapa se establecen los mecanismos refinados para el control del Backtracking, a través de la definición de apuntadores a la pila, conocidos como: CE

(*Control Environment*), BP (*Before Pointer*) y P (*Pointer*). Estos apuntadores permiten "recordar" los estados anteriores en el proceso de resolución. Estos estados deben ser recordados, ya que en caso de una falla en el procedimiento debe intentarse con otro espacio de búsqueda. Es importante resaltar que para nosotros, falla significa que le es imposible al compilador realizar la unificación entre dos términos. En caso de cubrir todo el espacio de búsqueda su respuesta es negativa a la consulta.

En una cuarta etapa se lleva acabo la inclusión de las características avanzadas de Prolog. Como por ejemplo, los cortes o *cuts* para mejorar el performance de un programa en Prolog, así como otras funciones como el caso de la operación "or". En esta etapa se define los diferentes mecanismos de procesamiento aritmético.

Nosotros utilizamos como base metodologica de diseño una variante de la propuesta de [AIK-1]. La variante consistió en fusionar las dos primeras etapas propuestas en [AIK-1], incluyendo adicionalmente, la información necesaria para definir la arquitectura de las clases Java. La tercera etapa fue orientada a la definición de los espacios de búsqueda pero utilizando clases en Java. En la última etapa, no avocamos a implementar las operaciones como "or" y las operaciones aritmeticas.

El concepto clave en la máquina de Warren es la definición de una metodología apropiada para manipular los términos de Prolog. Antes de continuar con la disertación es importante definir algunos conceptos propios del lenguaje de programación Prolog y su vinculación con la WAM.

Como se mencionó en el capítulo 2, Prolog está basado en la lógica clausal, que es a su vez un subconjunto de la lógica de primer orden. En la figura 2.4 se muestra la forma BNF de la sintaxis de Prolog.

En la WAM se define varios métodos para almacenar los términos en Prolog y manipular los términos. Por ejemplo, sea el término:

$$nuevo(A, viejo(A, Y), tiempo(Z, A)) \qquad (3.1)$$

En WAM, el término es visto como un conjunto de variables, constantes y funtores. Los funtores son un tipo de datos en Prolog, que está compuesto por un nombre y sus correspondientes argumentos. El número de argumentos de un functor se denomina aridad. Los argumentos del functor pueden ser terminos incluyendo functores.

Para cada uno de los términos almacenados en el programa se procede a realizar la descomposición del término en estructuras denominadas registros. Así pues, (3.1) se descompondría de la siguiente forma para ser almacenado en la pila de la WAM.

X1= nuevo (X2, X3, X4)
X2= A
X3= viejo (X2, X5)
X4= tempo (X6)
X5= Y
. *X6= Z*

Donde X1, X2,…,Xn, son variables libres o registros de variables libres, que permiten mantener la consistencia de las reglas y los términos asociados a los programas en Prolog. Luego de la descomposición de los términos, la máquina WAM debe proceder a almacenar los términos en una estructura similar a las que se muestra en la figura 3.1.

El procedimiento de almacenamiento del programa se lleva a cabo utilizando tres directivas básicas, *put_structure f/n, Xi; set_variable Xi* y *set_variable Xi*, donde *Xi* son variables libres.

Los términos de Prolog se cotejan (el proceso es conocido como *matching*) con la posición en memoria que reflejan las variables libres. Cada una de las variables libres está asociada a un registro. Aquí es importante recalcar que se puede realizar una analogia entre este tipo de registro y un registro de máquina tradicional. El registro es una localidad de memoria que permite almacenar estructuras de datos, y puede ser de 8, 16 o 32 bits como los registros tradicionales.

Dirección/Memoria	Etiqueta	Apuntador
0	Str	1
1	viejo/2	
2	REF	2
3	REF	3
4	Str	5
5	tiempo/2	
6	REF	6
7	REF	2
8	Str	9
9	nuevo/3	
10	REF	2
11	Str	0
12	Str	4

FIGURA 3.1. EJEMPLO DE ALMACENAMIENTO DE LA WAM

Existe un punto importante en el diseño de la WAM, es que se define una pequeña diferencia en la compilación de un programa y una pregunta. La pregunta siempre analizada de izquierda a derecha por ejemplo sea el programa.

nuevo(tiempo(A), espacio (S, tiempo(a)),S)

X1= nuevo (X2, X3, S)
X2= tiempo (A)
X3= espacio (a, X2)
X6= tempo (X7)
X7= a.

Ahora bien, si es una pregunta se compilaría de la siguiente forma:

nuevo (n, espacio (N,Z), tiempo(Z)

Seria entonces:

X3= espacio(N,Z)
X4= tiempo (Z)
X1= nuevo (n, X3, X4).

En el caso de una consulta o *query*, todo término adicional se incluirá en la pila originando o dando lugar a nuevas directivas para realizar la unificación con el programa ya almacenado en memoria, conocida como: *get-structure*, *unify-variable* y *unify-value*. Este proceso se lleva a cabo en tiempo de ejecución.

3.2. TRADUCCION DE PROLOG A JAVA.

La implementación de un lenguaje declarativo como Prolog en una estructura orientada a objetos, debe optimizar las búsquedas de los términos, es decir, maximizar la cantidad de términos procesados, ya que de continuar con una estrategia similar a la WAM tradicional implicaría un tiempo adicional no deseado. [CLP-2] [INT-1] [TAR-1].

Adicionalmente el principio de Warren, establece que:

> *"Los registros deben ser localizados de tal forma que se evite el movimiento innecesario de datos, así como minimizar el tamaño del código"*
>
> Warren

La aproximación orientada a objeto brinda varias posibilidades para construir modelos de sistemas complejos, pero solamente los lenguajes orientados a objeto con una semántica declarativa escrita son adecuados para el análisis lógico de los sistemas, incluyendo la programación de los mismos. [MOR-1]

Un punto importante en la programación de sistemas computacionales reactivos es que puedan trabajar en ambientes externos cambiantes. Morozov opina que bs lenguajes orientados a objeto permiten apuntar en esta dirección, ya que estos poseen modelos semánticos que son invariantes a conductas impredecibles desde un ambiente externo.

Otro aspecto de debate reside en el mantenimiento de la correctitud del sistema computacional. Morozov opina que "*todo el diseño de un agente puede ser comprendido utilizando solamente lenguajes orientados a objeto, incluyendo una descripción lógica de las operaciones y condiciones de los objetos*".

En la sección 3.3, se describirá la forma de implementar los mecanismos de Prolog basado en la WAM, en un programa realizado en Java, utilizando como punto de partida el principio de Warren. En la sección 3.4, se discutirá la arquitectura de agentes planteada en [DAV-1], y se mostrará nuestra aproximación para implementar las potencialidades de Prolog sobre Java como una plataforma para la programación de agentes, con una componente filosófica similar a la aproximación de Morozov.

3.3. DEFINICIÓN DE LOS MECANISMOS DE TRADUCCIÓN AUTOMÁTICA.

La idea general de un traductor de instrucciones de Prolog a Java, involucra la definición de la estructura de las clases que representen los predicados correspondientes, así como, los mecanismos que permitan implementar los procesos de búsqueda de soluciones propios de los lenguajes declarativos; aprovechando además las potencialidades de los lenguajes orientados a objetos.

Existen diferentes propuestas para la integración de lenguajes declarativos e imperativos. En el capítulo 2 se discutieron varias alternativas y los programas representativos de éstas. Nuestra propuesta se decanta por la alternativa de la traducción de Prolog a Java, pero incluyendo una innovación, en lugar de definir un conjunto de clases que implementen un compilador Prolog y que "consuma" las cláusulas y los términos del

programa; por lo que se procedió a definir el control sobre el espacio de búsqueda dentro de la misma clase que se asocia a un predicado. La razón de optar por una traducción de Prolog a Java, se sustenta en el punto de vista planteado por Morozov para la programación de agentes, tal como se discutió en la seccion 3.2.

El objetivo de la incorporación de mecanismos de control en las clases correspondientes, consiste en la independencia que puede brindar tanto a nivel de plataforma como de la estructura de una WAM particular. Adicionalmente, permite reducir el consumo de recursos, ya que la clase esta ajustada específicamente a la cantidad de cláusulas asociadas a un predicado. Ergo, será más eficiente, dado que las búsquedas se reducen a cambio de una serie de decisiones fijas.

En las próximas tres secciones discutiremos la información acerca de la estructura de las clases Java, justificación del por qué de este formato, definición de las clases funcionales, así como el procedimiento para la generación automática de las clases Java a partir de un programa en Prolog.

3.3.1. ESTRUCTURA DE LAS CLASES JAVA ASOCIADAS A PREDICADOS

En primer lugar, debemos definir la estructura de las clases que permitan llevar a cabo el proceso de integración de una plataforma netamente imperativa como lo es Java, con unos mecanismos y funciones que permitan implementar las facilidades de los lenguajes declarativos como Prolog.

La estructura de la clase diseñada utiliza como fundamento, la propuesta de [TAR-1][JIN-1], en cuanto a las características de las clases, en particular aquella que asocia el nombre de la clase al predicado, así como la información de la aridad especifica. Nuestra propuesta, en primer lugar define el nombre de la clase como el nombre del predicado más la aridad. La justificación de esta decisión se fundamenta, en que un mismo predicado puede tener diferentes cantidades de términos asociados, es decir, aridades diferentes.

Es oportuno, indicar que si asociamos a un predicado a una clase particular, se debe realizar un análisis del programa en Prolog en búsqueda de predicados de la misma estructura. Con un ejemplo se puede ver un poco más claro esta idea. Supóngase el siguiente código en Prolog,

padre(a,d).
hijo(a,f).
padre(X,G) :- hijo(X,H), hijo(G,H). (3.2)

Como puede observarse existen dos cláusulas asociadas al predicado padre/2 y una cláusula asociada al predicado hijo/2. A pesar que la última cláusula (padre(X,G):-hijo(X,H),hijo(G,H).) contiene el predicado hijo/2, que es una condición necesaria para inferir predicado padre/2, no se toma en cuenta para la generación de la clase Hijo_2. En este caso solo se deben generar dos clases: Padre_2 e Hijo_2. La asociación de predicados a cada clase para este caso, seria:

Padre_2: se asocia a las cláusulas:
padre(a,d).
padre(X,G) :- hijo(X,H), hijo(G,H).

Hijo_2: se asocia a las cláusulas:
hijo(a,f).

Bajo la consideración anterior, se puede inferir que la estructura de las clases involucra la agrupación de cláusulas cuyos consecuentes sean del mismo formato del predicado. Esto conduce adicionalmente a definir la cantidad de cláusulas, -en este caso las llamaremos reglas-, así como los átomos asociados al predicado.

Este proceso debe incluir la definición de dos atributos de las clases. El primero, asociado con la cantidad de reglas cuya cabecera sea el mismo predicado analizado. El segundo, almacena la cantidad de átomos del tipo predicado. Estos valores se almacenan en dos atributos conocidos como: ***numberRules*** y ***numberAtoms***, respectivamente.

La asignación de nombre a las clases tiene correspondencia con la directivas *put_structure* y *get_structure* de la WAM [AIK-1]. Las directivas *put_structure* y *get_structure*, manipulan los predicados de forma tal, que toman el nombre del predicado y lo almacenan en la estructura conocida como *head*, así como la realización del apartado de espacio en memoria para almacenar la cantidad de términos que indica la aridad.

En la estructura de los atributos de la clase, se procedió a definir un campo vinculado con el almacenamiento de los términos *grounded* de forma particular. La idea subyacente, es la realización de una búsqueda más expedita de las consultas hechas a las clases Java. Al igual que en la WAM, cuando hay términos *grounded* las localidades de memoria se inicializan con los valores del termino *grounded*: nombre del termino y la aridad, junto con los correspondientes términos de sus argumentos; a través de etiquetas invariables mientras permanezca el programa en memoria principal. Ergo, las consultas a los registros que tienen términos *grounded* o constantes son inmediatas en la WAM. Se

excluyen los términos no *grounded* de los atributos de la clase Java, ya que ellos pueden poseer una o varias variables, por lo cual, tienen una influencia particular en espacios de búsqueda, y deberán ser tratados de forma diferente.

Los términos *grounded* se almacenan en una lista enlazada, para ser más específicos se almacenan solo los subtérminos del predicado en análisis. La lista de términos *grounded* recibe el nombre de ***facts***. Entonces, la estructura inicial de la clase será:

```
class Predicado_aridad{
        ListLinked facts = new ListLinked();
        numberRules = int;
        numberAtoms = int;
}
```

Se debe incluir en la estructura de la clase, métodos que permitan obtener la información de los atributos propios de la clase definida. Los métodos definidos para tal razón se muestran en la Tabla 3.a, así como la descripción de los mismos.

Método	Descripción
public void addElement (Term, Term, ..., Term)	**Este método permite incluir nuevos elementos a la Clase. Deben indicarse los términos asociados al predicado**
public void removeElement (Term, Term, ..., Term)	**Este método permite eliminar elementos a la Clase.**
public int getArity ()	**Devuelve el valor de la aridad asociada a la Clase**
public int getNumberRules()	**Devuelve la cantidad de Reglas incluidas en la Clase correspondiente**
public int getNumberFacts()	**Devuelve la cantidad de átomos incluidos en la Clase correspondiente**
public LinkedList getFactsList ()	**Devuelve todos los subtérminos de todos los átomos definidos para la Clase.**
public int getNumberTerms ()	**Devuelve la cantidad de Términos almacenados en la lista enlazada**
public int getNumberElements()	**Devuelve la cantidad de átomos almacenados**
public Term getTerm (int)	**Devuelve el i-esimo término almacenado en la lista.**

TABLA 3.A. METODOS BÁSICOS DE LA CLASE JAVA

La estructura de la clase incluidos los métodos básicos será:

```
class Predicado_aridad{
        ListLinked facts = new ListLinked();
        numberRules = int;
        numberAtoms = int;

        public void addElement (Term, Term, ..., Term) {...}
        public void removeElement (Term, Term, ..., Term) {...}
        public int getArity () {...}
        public int getNumberRules () {...}
        public int getNumberAtoms () {...}
        public LinkedList getFactsList () {...}
        public int getNumberElements () {...}
        public int getNumberTerms() {...}
        public Term getTerm(int) {...}
}
```

En este punto, es importante mencionar que en la definición de los mecanismos de traducción en nuestra propuesta como se pude ver, se hace una diferencia de las cláusulas. Las cláusulas se dividen en dos grupos. El primer grupo corresponde a las cláusulas que en Prolog reciben el nombre de hechos (*facts* en inglés). El segundo grupo corresponde a las cláusulas llamadas reglas. La idea detrás de esta diferenciación estriba en la posibilidad de permitir la incorporación dinámica de cláusulas a los objetos Java, previamente creados a partir del programa en Prolog. En nuestra propuesta, se diseñaron los mecanismos para incorporar y desincorporar cláusulas cuya estructura sean hechos. Se implementaron dos métodos para tal fin: *addElement* y *removeElement*.

Posteriormente, se debe definir el método de búsqueda de términos *grounded*, es decir, solo si los términos no contienen variables. Si existe algún átomo que corresponda con este tipo de consulta, el programa deberá responder de forma afirmativa. La aseveración anterior implica la definición de un método que permita implementar este mecanismo. El método se denomina *searchT*, tal como se muestra a continuación;

```
public boolean searchT(Term, Term, ..., Term, int)
```

Donde *Term* representa a los términos asociados a los términos del predicado vinculado a la clase. La información de los términos se define como una clase funcional que se describirá en secciones posteriores, y que es imprescindible para la implementación de los mecanismos de los lenguajes declarativos en una plataforma orientada a objetos.

Por ejemplo, supóngase el programa (3.2) y que se realiza la consulta *?padre(a,d)*. La devolución o respuesta a la consulta debe ser una respuesta afirmativa, al igual que en un compilador tradicional Prolog.

3.3.1.1. IMPLEMENTACIÓN DE LA UNIFICACIÓN DE PROLOG EN LAS CLASES JAVA.

Ahora bien, una vez resuelta la búsqueda de términos *grounded*, debemos resolver las consultas que abarque variables, por ejemplo, sea la consulta *?padre(X,Y)*. Una respuesta a una consulta de este tipo al programa (3.2), seria

$X = a,\ Y = f.$

Para la consecución de este objetivo, es necesaria la definición del mecanismo de unificación, para llevar a cabo esta operación. En primer lugar, se debe implementar un algoritmo de unificación y los parámetros requeridos para su funcionamiento en Java. El algoritmo utilizado consiste en una modificación del algoritmo de Robinson descrito en el capítulo 2.

En este punto, existían dos posibilidades de diseño. La primera consistía en implementar el algoritmo de Robinson como un conjunto de métodos de la clase Java asociada a un predicado. La segunda posibilidad como una clase independiente a las clases Java. Nuestra propuesta se decantó por la segunda alternativa, ya que permite que el código de la clase Java sea menor y más legible.

Se creó una clase *UnifTerm.java,* en la cual se incluye el algoritmo de unificación, junto con la generación del *Unificador Más General*, que consiste de un vector de términos en el cual se asocia una variable libre con un término, esto corresponde a la vector *fi* de la sección 2.8.4.

Adicionalmente, se define los mecanismos de construcción de términos a partir del *Unificador Más General*, de forma tal de obtener átomos de otros átomos con variables libres, en nuestro caso las denominaremos *Hi*. Esta acción de construcción corresponde al concepto conocido como sustitución [HOG-1]. Estas variables libres se asocian con la información de los registros de la WAM.

El método que permite realizar la unificación se denomina *unify*, en sus dos versiones:

public Vector unify (Vector , Vector , int)

public Vector unify (Vector , Vector)

El método *unify,* permite implementar el algoritmo de unificación de Robinson discutido en la sección 2.8.4. Los dos primeros argumentos corresponden a las estructuras donde se

almacenan los subtérminos de los dos predicados que se desean unificar. El tercer argumento es tentativo y corresponde a la aridad del término. El resultado del método *unify*, en caso de ser exitoso el proceso de unificación, corresponderá al *unificador más general* y que se almacenará como un vector de términos.

Los métodos encargados de llevar a cabo la sustitución se denominan *buildSus*. Se discutirá con mayor detalle en la sección correspondiente a las clases funcionales.

Retomando la discusión referente al manejo de variables, se define el método de búsqueda de términos, cuya consulta abarque algún tipo de variable. El método se denomina *searchTCV,* tal como se muestra a continuación;

public boolean searchTCV(Term, Term, ..., Term, int)

La estructura del método s*earchTCV*, tiene como parámetros de entrada una cantidad de términos asociados a la cantidad de subtérminos del predicado, de forma idéntica al método *searchT*.

Para llevar a cabo el proceso de unificación se definieron tres vectores: *varbl, argum y argUf*. El primero consiste en todos los términos asociados a las variables libres, como en el caso de un compilador de Prolog tradicional. Podemos ver este concepto, como una estructura temporal que permita almacenar el estado de una búsqueda en un momento particular. Este tipo de mecanismo se asocia con las directivas definidas en WAM como *get_unify*, así como el apuntador denominado *pointer*. El segundo vector permite almacenar la información correspondiente a los términos de entrada, en este caso constituyen las variables que forman la consulta. El tercero constituye el compendio de los valores temporales del proceso de unificación. Este vector contiene las variables asociadas a los subtérminos del predicado sobre el cual se está ejecutando la unificación.

Para distinguir los términos manipulados por cada uno de los vectores asociados a la unificación, se utiliza la nomenclatura mostrada en la tabla 3.B.

Vector	Variable asociada	Descripción de la variable
argum	**Xi**	**Términos de entrada**
argUf	**Ti**	**Términos temporales**
varbl	**Hi**	**Términos asociados a las variables libres**

TABLA 3.B. VECTORES DE CONTROL DE LA CLASE JAVA Y VARIABLES ASOCIADAS

Para entender un poco mejor la utilización de los vectores *argum, argUf y varbl*, veamos un ejemplo. Supongase que se tiene un predicado de la forma *a(Vaq1, pr_b),* y se realiza

una consulta del tipo *?a(pr_a,pr_b)*. Entonces, los vectores contendrán los siguientes valores antes de la unificación:

argum = { pr_a, pr_b}
argUf = { H1, pr_b}
varbl = { H1, H2}

Una vez realizado el proceso de unificación los vectores contendran los siguientes valores:

argum = { pr_a, pr_b}
argUf = { pr_a, pr_b}
varbl = { H1, H2} *donde H1 = pr_a y H2 = pr_b*

Podemos observar como las variables *Xi* asociadas al vector *argum*, permanecen invariables durante el proceso de unificación. Las variables *Ti* del vector *argUf*, se inicializan de acuerdo a la definición presente en la clase - que fue tomada de un predicado en Prolog -, excepto que las variables del predicado ahora se expresan como variables libres *Hi*. Las variables *Ti* se modifican según el valor aportado por las variables libres asociadas a las variables *Hi* del vector *varbl* una vez concluido el proceso de unificación.

?

En el momento de llevar a cabo el proceso de generación de las clases por medio de los programas previamente discutidos, se utiliza un método particular de *parsing*. Los átomos que se almacenarán dentro de este método, deberán sustituir las variables que lo definen y sustituirse por variables libres en nuestro caso por términos *Hi*. Por ejemplo, sea la regla 3 del programa 3.2 que expresa que.

padre(X,G):

El programa generador la convertiría en:

padre(H1,H2).

Entonces se almacenará en las variables temporales con los valores de los términos H1 y H2, es decir,

```
public boolean SearchTCV(Term X1, Term X2, 1){
        …
        T1.asigvalue(H1);
        T2.asigvalue(H2);
        …
}                                                   (3.3)
```

El método *asigValue*, permite asignarle el valor de un término a otro término. En el caso de (3.3) se le asigna el valor de un término *Hi* a un término *Ti*. El método *asigValue* pertenece a la clase *Term,* que será discutida en la sección 3.3.4.1.

Se puede observar que el proceso anterior, es similar al que se lleva a cabo cuando se genera la estructura de pila en un WAM. Una vez, que se introduce el programa o la consulta la WAM a través de una serie de directivas, se almacenan los términos asociados a las diferentes cláusulas en la *head,* como se discutió en el ítem 3.2.

En el caso de la WAM el predicado,

padre(X,G).

estará asociado con una estructura similar a la siguiente:

Dirección/Memoria	Etiqueta	Apuntador
0	**STR**	**1**
1	**padre/2**	
2	**REF**	**2**
3	**REF**	**3**

FIGURA 3.2. REPRESENTACIÓN DE UN PREDICADO EN WAM

Puede observarse que es irrelevante el nombre de las variables, ya que para WAM constituyen apuntadores específicos a registros en la memoria de la máquina. Una situación similar sucede en nuestra propuesta, en la cual los registros de Prolog se substituyen por términos denominados *Hi*.

Una vez definidos los vectores correspondientes se procede a realizar la unificación entre los vectores correspondientes a los términos *Xi* y *Ti*. Si el algoritmo de unificación puede lograr su cometido, se procede a la construcción de los términos que será almacenados en unas estructuras denominadas *termAi* y, que representan el resultado de la consulta. Continuando con el ejemplo 3.3, se podría representar esta situación con el siguiente pseudocódigo,

```
mGUnf = (UnifyTerm) unify(argum, argUf, arity);

if (unificación){
        ...
        termA1 = (UnifTerm).buildSus(T1, mGUnf);
        termA2 = (UnifTerm).buildSus(T2,mGUnf);
        ...
}                                                    (3.4)
```

donde *mGUnf*, es el unificador más general [HOG-1].

El *unificador más general* ó *mGUnf* se obtiene como respuesta a la aplicación del método *unify* presente en la clase *UnifTerm*, como se mencionó anteriormente. La construcción de términos utilizando el procedimiento de sustitución, se lleva a cabo usando el método *buildSus* definido dentro de la clase *UnifTerm*. Esto se discutirá en la sección 3.3.4.

En (3.4) se puede observar que se requiere definir un conjunto de atributos que permitan almacenar las respuestas a las consultas, es por ello, que se define *n* cantidad de términos como atributos, donde *n* es la aridad del predicado que estamos tratando en la clase. Entonces nuestros atributos se ampliarían a,

```
class Predicado_aridad{
        ListLinked facts = new ListLinked();
        numberRules = int;
        numberAtoms = int;
        Term termA1;            //termino_1
        Term termA2;            //termino_2
        ...
        Term termAn;            //termino_n
                                //donde n es la aridad
}
```

Resulta conveniente que la salida de nuestra clase, pueda tener al menos un contacto con la salida estándar, es por ello, que se procedió a implementar un método que permitiera imprimir en pantalla los valores provenientes de la respuesta a una consulta determinada. Bajo nuestro esquema, las respuestas a las consultas se almacenan en los términos especiales –que denominados *termAi* -, por lo tanto, utilizamos un método definido en la clase *Term* que permite imprimir en pantalla el valor de los términos *termAi*.

El método de impresión de la clase se denomina:

```
public String printAnswer(Term termIn, Term termExt)
```

donde *termIn* es el término a imprimir y *termExt* retiene el valor inicial del termino antes de cualquier procesamiento.

3.3.1.2. SIMULANDO LA ESTRATEGIA DE SELECCIÓN DE CLAUSULAS DE PROLOG.

Ahora bien, hasta el momento se ha discutido la estructura de la clase en Java cuando se incluyen términos *grounded* y átomos en general. Debemos ahora definir la estructura que debe implementarse para tratar con las reglas. Nuestra propuesta incluye un nuevo método denominado:

public boolean searchRule(Term, Term, ..., Term, int)

que tiene como parámetros los subtérminos del término a consultar, de forma similar al método s*earchT*.

En el método *searchRule* residen agrupadas, todas las reglas asociadas al predicado que se encuentran dentro del programa Prolog sujeto a análisis. Es importante definir la estructura interna de cada regla para que se lleve a cabo el proceso de inferencia como en el caso de los lenguajes declarativos tradicionales.

En primer lugar el método utiliza los mismos tres vectores discutidos en el caso del método *searchTCV*. Lo cual implica que las reglas también sufren la sustitución de las variables incluidas dentro del programa Prolog, por variables libres que en nuestro caso son términos del tipo *Hi*, como se indicó anteriormente.

En cada una de las reglas se trata de unificar las variables de entrada –como mencionamos anteriormente, denominadas *Xi* - con respecto a las variables temporales –terminos *Ti* -, este proceso permite determinar si se aplica una regla o no. En caso de que la unificación fuese efectiva, se debe aplicar un proceso de búsqueda para cada uno de los términos que conformen las condiciones de la cláusula asociada a la regla. Como se discutirá posteriormente el control asociado a la clase está determinado por el método *searchT*.

[TAR-1][COD-1][CAL-1] proponen una estructura similar a un “case” de un lenguaje como C. Nuestra propuesta incluye una generalización, utilizando estructuras simples y encadenadas como lo son: *while-if*.

La estructura *while* permite construir el espacio de búsqueda similar a una estructura de un árbol, y definir la búsqueda de soluciones dentro de esta estructura, bajo la estrategia *top-down* y primero en profundidad. En Prolog se conoce esta estrategia como encadenamiento hacia atrás o *backward chaining.*

Se establece la estructura *while-if*, una por cada uno de los términos existentes como condición para verificar la asertividad de una regla. A continuación se muestra la estructura del *while*:

while (j1=0 ó j1< predicado_regla.getNumberTerms())

donde *j1* es una variable entera y *predicado_regla* es una clase asocida al predicado que es una condición de la regla analizada. El metodo *getNumberTerms* devuelve la cantidad de terminos *grounded* definidos en la clase *predicado_regla.* La variable *j1*, se utiliza para controlar el acceso a las estructuras *if*. Como puede verse, tenemos dos sentencias disjuntas. La segunda de las sentencias permite el recorrido del árbol de búsqueda, a traves de los términos definidos en la clase *predicado_regla*. La primera de las sentencias garantiza que en el caso de la consulta se acceda al menos una vez a las sentencias *if*. De no existir esta última condición, y se diese el caso de que la clase Java no poseyera términos *grounded*, la consulta nunca accederia a las sentencias *if*, lo cual eliminaria un nodo o conjunto de éstos en el árbol de búsqueda, lo cual es una condición indeseada.

La instrucción *if* permite realizar búsquedas dentro de la clase o invocar predicados de otras clases, así podemos definir la búsqueda de la siguiente forma:

if (predicado_regla.searchT (term, ..., term, int) = true)

La instrucción anterior especifica que si un predicado de la regla ha sido resuelto por medio del método *searchT* - lo cual implica que la instrucción *if* es cierta -, y por lo tanto obtenemos la respuesta a la consulta que solicitamos. Por último, se utiliza el mismo procedimiento de impresión de resultados que el aplicado al método *searchTCV*, es decir,

```
mGUnf = (UnifyTerm) unify(argum, argUf, arity);
...
termA1 = (UnifTerm).buildSus(T1,mGUnf);
termA2 = (UnifTerm).buildSus(T2,mGUnf);
...
termAn = (UnifTerm).buildSus(Tn,mGUnf);     //donde n es la aridad del término
...
```

Para aclarar un poco mejor el funcionamiento de la estructura *while-if*, veamos un ejemplo en Prolog:

```
inicio(0).
inicio(X):- Z is (X-1), inicio(Z).                                   (3.5)
```

Supóngase el programa (3.5), en el que la segunda cláusula hace un llamado a si misma, pero con una variable libre diferente, entonces se da origen a la búsqueda de términos

que puedan satisfacer dicha regla. La especificación de la estructura se despliega a continuación:

```
while (j1=0 ó j1< getNumberTerms())    {
        …
        if( searchT(term, int) = true){
                …
                retorno = true;
                return retorno;
                …
        }
        j1++;
}
```

Puede observarse que no se incluye un ciclo para el término *Z is (X-1)*, ya que debería omitirse todo aquello que incluya operaciones básicas como la suma, la resta, entre otras. Nuestra propuesta define una clase funcional denominado *Operators* donde se incluyen las operaciones elementales. Las clases funcionales se describen como más especificidad en la sección 3.3.4.

Hemos visto la información cuando se procesan reglas, cuyos predicados se encuentran dentro de las condiciones de la cláusula, pero se omite la situación en la cual se hace un llamado a un predicado que es diferente al asociado a la clase actual. Para realizar este tipo de tarea, se requiere la instanciación de las clases Java asociadas a los predicados correspondientes.

Por ejemplo supóngase que se incluye al programa (3.2), una cuarta cláusula que corresponda a la siguiente definición:

hermano(X,G) :- hijo(H,X), hijo(H,G).

a la cual, se le aplican los criterios de traducción previamente discutidos. Ergo, el predicado *hermano/2* estará asociado a la clase *Hermano_2.java*, que hace un llamado al predicado *hijo/2*, por lo que se requiere instanciar un objeto del tipo *Hijo_2*, de la forma siguiente:

```
Hijo_2 hijo_2i = new Hijo_2();
```

La estructura de búsqueda para la regla "hermano(X,G) :- hijo(H,X), hijo(H,G)", se definirá entonces como:

```
while(i1=0 ó i1< hijo_2i.getNumberTerms()).    {
        …
        if(hijo_2i.searchT(term, term, int)=true){
                if(hijo_2i.searchT(term, term, int)=true){
                                …
                                retorno = true;
                                …
                                return retorno;
                }
        }
        i1++;
}                                                                    (3.6)
```

Ergo, por cada llamada a un tipo predicado en la regla, se crea una estructura *while* y por cada una de las apariciones de un predicado se crea una estructura *if*. Por ejemplo, en el caso de regla "hermano(X,G) :- hijo(H,X), hijo(H,G)". Tenemos un ciclo *while*, debido al predicado *hijo/2;* y dos estructuras *if,* por los términos hijo(H,X) y hijo(H,G).

Para entender un poco mejor el funcionamiento de la estructura *while-if*, veamos una corrida en frio del programa siguiente,

```
hijo(edgar, leo).
hijo(edgar, carlos).
hijo(edgar, jhon).
hijo(jhon, jhonjr).
abuelo(H,G) :- hijo(H,X), hijo(X,G).                                 (3.7)
```

al cual queremos consultar quien es el nieto de 'edgar', es decir, *?abuelo(edgar,C)*.

En primer lugar, como se discutió anteriormente, se crean dos archivos *Abuelo_2.java* e *Hijo_2.java.* Los atributos de la clase *Hijo_2.java* serán,

```
numberAtoms = 4;
numberRules = 0;
arity = 2;
termA1 = new Term();
termA2 = new Term();

facts.add(new Term ("edgar",0));
facts.add(new Term ("leo",0));
facts.add(new Term ("edgar",0));
facts.add(new Term ("carlos",0));
facts.add(new Term ("edgar",0));
facts.add(new Term ("jhon",0));
facts.add(new Term ("jhon",0));
facts.add(new Term ("jhonjr",0));
```

En el caso de esta clase, como solo tenemos términos *grounded*, no se crea estructura *while-if* alguna. En el caso de la clase *Abuelo_2.java*, los atributos serán:

```
numberAtoms = 0;
numberRules = 1;
arity = 2;
termA1 = new Term();
termA2 = new Term();
```

adicionalmente se definirá una estructura de la forma, donde se incluye la estructura while-if y la verificación de la realización del proceso de unificación.

```
H1.changeTerm("H1"+CBackT,0);
H2.changeTerm("H2"+CBackT,0);
H3.changeTerm("H3"+CBackT,0);
…
//Resolucion.
…
if (unificación){
        while(i1==0 || i1 < hijo_2i.getNumberTerms()){
                …
                if (hijo_2i.searchT(H1,H3,CBackT)== true){
                        …
                        if (hijo_2i.searchT(H3,H2,CBackT)== true){
                                …
                                retorno = true;
                                (Seudocodigo) termA1 = buildSusTV(T1, varbl);
                                (Seudocodigo) termA2 = buildSusTV(T2, varbl);
                                return retorno;
                        }
                }
                i1++;
        } //fin de ciclo i1
}                                                                   (3.8)
```

Una vez se realiza la consulta *?abuelo(edgar,C),* se lleva a cabo en primer lugar un proceso de verificación para saber si existe un término que pueda ser unificado directamente con *abuelo(edgar,C).* En el programa (3.7) no existe un término *grounded* asociado al predicado *abuelo/2*, por lo cual, el programa procede a ejecutar la estructura asociada a la regla *abuelo(H,G) :- hijo(H,X), hijo(X,G)*, que se muestra en (3.8).

Se inicializan las variables libres *Hi*, en este caso *H1, H2* y *H3*. Existen tres variables libres debido a la forma como se estructura la regla, es decir, *abuelo(H1,H2):-hijo(H1,H3),hijo(H3,H2)*. El siguiente paso consiste en realizar el proceso de unificación entre *abuelo(edgar,C)* y *abuelo(H1,H2)*. Lo anterior permite determinar si puede o no ser aplicada una regla. En el caso del ejemplo, se puede observar que la sustitución debe ser *H1/edgar* y *H2/C*.

Una vez se verifica que se puede aplicar la regla, por medio de la instrucción *if(unificación)* en el ejemplo (3.8), el programa ingresa al ciclo *while*, el cual permite "recorrer" todos los términos grounded de *hijo/2.* Es importante mencionar, que nuestra propuesta mantiene la misma estrategia de Prolog, ya que la primera cláusula que aparece en el programa, es la primera en ser probada.

El primer término asociado a *hijo(edgar,C)* en el programa (3.7) es *hijo(edgar, leo).* Este término se asocia a la instrucción *if(hijo_2i.searchT(H1,H3,CBackT)==true),* donde *H1/edgar* y *H3/leo*. En la clase *Hijo_2.java* existe el término *hijo(edgar,leo)*, por lo cual, la instrucción *if* anterior tendrá un valor igual a *true*. Se procede luego, a ejecutar la siguiente instrucción *if(hijo_2i.searchT(H3,H2,CBackT)==true),* donde *H3/leo* y *H2/C.* Ahora, al realizar la consulta a la clase *Hijo_2.java,* programa no encuentra términos que satisfagan la consulta, por lo que la segunda instrucción *if* tendrá un valor igual a *false.* El programa ahora retorna el control al ciclo *while,* y se intenta ahora con el siguiente término, en este caso *hijo(edgar, carlos).* Se sigue el mismo procedimiento que en el caso *de hijo(edgar, leo),* y con idéntico resultado, ya que no existe un término que pueda ser unificado con *hijo(carlos, C).* Por último, se prueba con el término *hijo(edgar,jhon),* pero a diferencia de los dos casos anteriores, la segunda instrucción *if* tiene un valor igual a *true;* debido que puede validar la existencia del término *hijo(jhon,jhonjr)* en la clase *Hijo_2.java*. Ergo, asigna los términos *termA1* y *termA2* de la clase *Abuelo_2.java*, con los valores que arrojan *T1* y *T2,* luego de realizar la sustitución con los valores que tienen las variables *H1* y *H2*.

En la figura 3.3.1 se puede observar el grafico de búsqueda SLD de programa en Prolog (3.7). Se muestra ademas, el recorrido del programa en Java haciendo hincapié en los metodos de control disenados. En esta seccion se omite la explicación del funcionamiento de los métodos *searchRule, searchT, searchTCV* y *searchRuleVar*, así como de la variable *CBackT* que se discutirán en la sección 3.3.1.

?

En el caso de traducciones como la presentada en (3.6), podria representarse como una estructura *case* o *if* solamente, en lugar de una estructura *while-if*, es decir,

```
if (hijo_2i.searchT(H3,H1,CBackT)== true){
        ...
        if (hijo_2i.searchT(H3,H2,CBackT)== true){
                ...
                retorno = true;
                ...
                return retorno;
        }
}
```

La estructura *while-if* se debe utilizar imprescindiblemente en aquellos casos en los que se requiera recursividad, como en el caso del programa (3.5), en todos los demás casos puede utilizarse solo estructuras del tipo *if*.

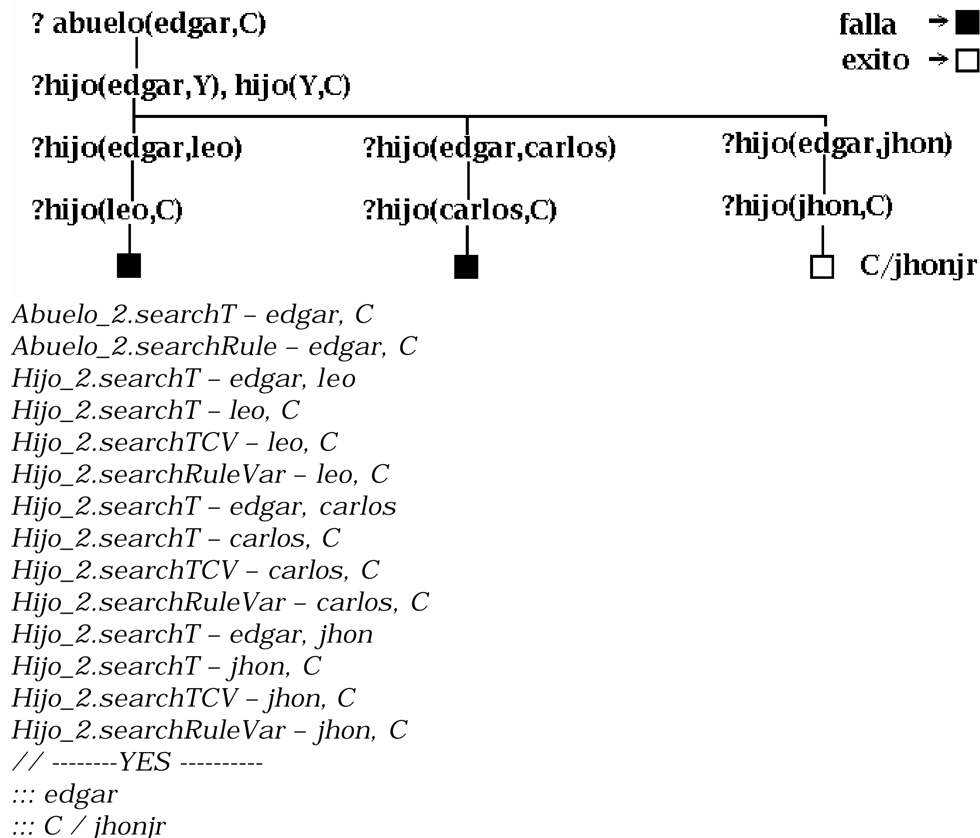

FIGURA 3.3.1. ARBOL DE BUSQUEDA Y METODOS DE CONTROL.

En nuestra propuesta, las estructuras *while-if* se generan en todas las traducciones de Prolog a clases Java. Una razón por la cual en el código se mantuvo la estructura de *while-if* para todas las cláusulas en lugar "adecuar" la estructura while-if o if, según fuese el caso; fue mantener una estructura general para todas las clases, de forma que permita tener un código flexible y genérico. Otra razón para generalizar el uso de las estructuras *while-if*, es minimizar el tiempo de compilación, ya que requería mayor tiempo analizar la existencia de recursividad de una determinada regla.

?

El ultimo método a tratar, consiste en aquel que permite devolver valores no *grounded*, que lo denominamos *searchRuleVar*. Posee la misma estructura básica del método *searchTCV*, pero solo devuelve valores de las variables *Hi*.

A continuación se presenta la estructura de la clase especificada.

```
public boolean searchRuleVar(Term, Term, …, Term, int)
```

Para garantizar un criterio de estandarización en la totalidad de la traducción, se estipuló que toda instancia de una clase-predicado que se lleve a cabo en otra, deberá comenzar con el nombre del predicado seguida por la aridad asociada y terminar en *i*. Por ejemplo,

```
Hijo_2 hijo_2i = new Hijo_2();
Padre_2 padre_2i = new Padre_2();
```

En la figura 3.3 se presenta un resumen de la estructura de las clases en Java asociadas a predicados, discutidas en las secciones previas.

```
class Predicado_aridad{

        //atributos
        ListLinked facts = new ListLinked();
        numberRules = int;
        numberAtoms = int;
        Term termA1;            //termino_1
        …
        Term termAn;            //termino_n

        //métodos básicos
        public void addElement (Term, Term, …, Term) {…}
        public void removeElement (Term, Term, …, Term) {…}
        public int getArity () {…}
        public int getNumberRules () {…}
        public int getNumberAtoms () {…}
        public LinkedList getFactsList () {…}
        public int getNumberElements () {…}
        public int getNumberTerms() {…}
        public Term getTerm(int) {…}

        //metodos principales de control
        public boolean searchT(Term, Term, …, Term, int) {…}
        public boolean searchTCV(Term, Term, …, Term, int) {…}
        public boolean searchRule(Term, Term, …, Term, int) {…}
        public boolean searchRuleVar(Term, Term, …, Term, int) {…}
}
```

FIGURA 3.3.2. ESQUELETO DE LAS CLASES JAVA ASOCIADAS A PREDICADOS

3.3.1. DEFINICIÓN DEL CONTROL EN LAS CLASES JAVA.

A continuación discutiremos el proceso de resolución en Prolog, y nuestra propuesta de implementación de éste. En el capítulo 2 se describió de modo general el concepto de resolución.

El algoritmo de SLD, cuyo nombre proviene de las palabras en inglés: *Selection, Linear and Definite*. La selección involucra la escogencia de un literal particular a partir del conjunto de reglas definidas, y que se lleva a cabo en cada paso de computación. La linealidad hace referencia al hecho, de que en cada paso se utiliza el más reciente resolvente como pregunta siguiente. Por último, se supone que todas las cláusulas son definidas.

Dicha estrategia del algoritmo de SLD, puede presentar varias alternativas de cómputo, determinando la forma de sus exploraciones y por lo tanto, determinando el tiempo consumido para encontrar la respuesta, así como la eficiencia de la misma.

La realización de la búsqueda entre las diferentes alternativas, es una característica de los formalismos relacionales – tales como los lenguajes lógicos -, que los distingue de los formalismos funcionales. Existe, por supuesto, diferentes formas de búsqueda - equivalentemente a la construcción de un árbol de búsqueda -. Las opciones incluyen búsqueda serial ó paralela y/o búsqueda *top-down* ó *bottom-up*.

La estrategia de búsqueda estándar usada por un programa lógico interpretado por una máquina con un simple procesador es una búsqueda primero en profundidad, secuencial, top-down con backtracking. En la figura 3.4, se muestra cual seria el rastro de búsqueda de un algoritmo de control con la propiedad de búsqueda primero en profundidad, secuencial, top-down con backtracking. Esta estrategia es la que implementa el algoritmo de control para procesar la búsqueda de soluciones.

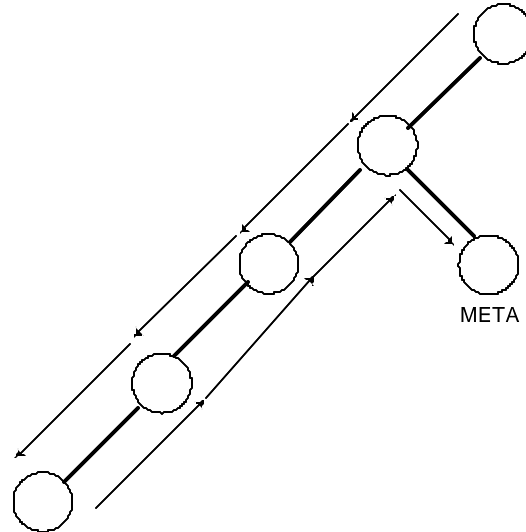

FIGURA 3.4. ESQUEMA DE BÚSQUEDA DEL ALGORITMO SLD.

Como se indicó en la sección previa, nuestra principal innovación consiste en la inclusión del control declarativo en las clases Java asociadas a predicados de Prolog. Es importante mencionar, que de aquí en adelante utilizaremos la frase *"control declarativo"* para hacer referencia a la implementación de los mecanismos propios de los lenguajes declarativos como la resolución, en la definición de clases de Java.

A continuación describiremos el algoritmo de control para la especificación de la clase antes mencionada que involucra implementar el control utilizado en los lenguajes declarativos. Dado que una implementación de forma estricta e idéntica a Prolog en un lenguaje como Java daria lugar a estructuras que consumirian recursos computacion, debido a que en primer lugar no tendríamos una estructura de registros optimizada para la plataforma, y en segundo lugar seria ineficiente la existencia de una pila única o *head* en Java. Lo anterior conlleva a utilizar estrategias y métodos para la definición del control de flujo de las operaciones propias de Prolog, pero ahora en las clases definidas en Java.

Otra punto importante en cuanto al manejo de una estructura de este tipo, radica que al igual que en un programa Prolog, una consulta desencadena una serie de llamados a predicados de forma automática. En el caso de Prolog el programa principal siempre tiene el control, ya que todos los términos del programa están almacenados en la estructura definida por la WAM, pero la diferencia principal en nuestra propuesta estriba en que la clase que se instancia desde la clase de consulta desencadena el control de la búsqueda.

Una ventaja radical de este tipo de arquitectura, es que se puede realizar de forma independiente la modificación de cada uno de los predicados sin afectar la totalidad del programa, es decir, podemos realizar una compilación selectiva de predicados. Por

ejemplo, si de (3.2) decidimos cambiar la información de la tercera cláusula, agregando la instrucción *not(F=G),* solo se requiere la generación de la clase Padre_2. En el caso de Prolog debemos recargar todo el programa, ya que los predicados se cargan en memoria, esto se lleva a cabo con consult/2.

Dada la estructura discutida en la sección anterior, procedemos a incluir los mecanismos de control en la clase Java. La inclusión está determinada por el orden de las llamadas de cada uno de los métodos asociados a la búsqueda, a saber,

public boolean searchT(Term, Term, …, Term, int)
public boolean searchTCV(Term, Term, …, Term, int)
public boolean searchRule(Term, Term, …, Term, int)
public boolean searchRuleVar(Term, Term, …, Term, int)

En primer lugar, el inicio del ciclo de consulta está determinado por la existencia de términos *grounded*, es por ello que el primer método tratado en la consulta es *searchT*. Si se comprueba la existencia del término, devuelve una respuesta afirmativa, y es almacenada la respuesta en los atributos de la clase correspondientes para eso fines. En caso que la consulta sea negativa, se requiere verificar si hay o no reglas que tratar.

Si en la clase no hay reglas que procesar, se debe verificar si hay términos con variables. En caso afirmativo se procede a llamar el método *searchTCV.* En la figura 3.5 se muestra el caso del control cuando no se analizan las reglas.

En el caso de que existan reglas, se debe realizar un análisis de cada una de ellas. Se procede a realizar el llamado al método *searchRule*. En el método *searchRule* residen todas las reglas asociadas al predicado sujeto a análisis. La selección de la ejecución de una regla está dado por la capacidad de unificar o no el término a consultar. Adicionalmente, se debe indicar que en el método *searchRule*, se puede realizar un llamado al método *searchTCV*, en caso de que existan términos con variables en la especificación de la clase. En la figura 3.6 se resume el control cuando existen reglas en la clase.

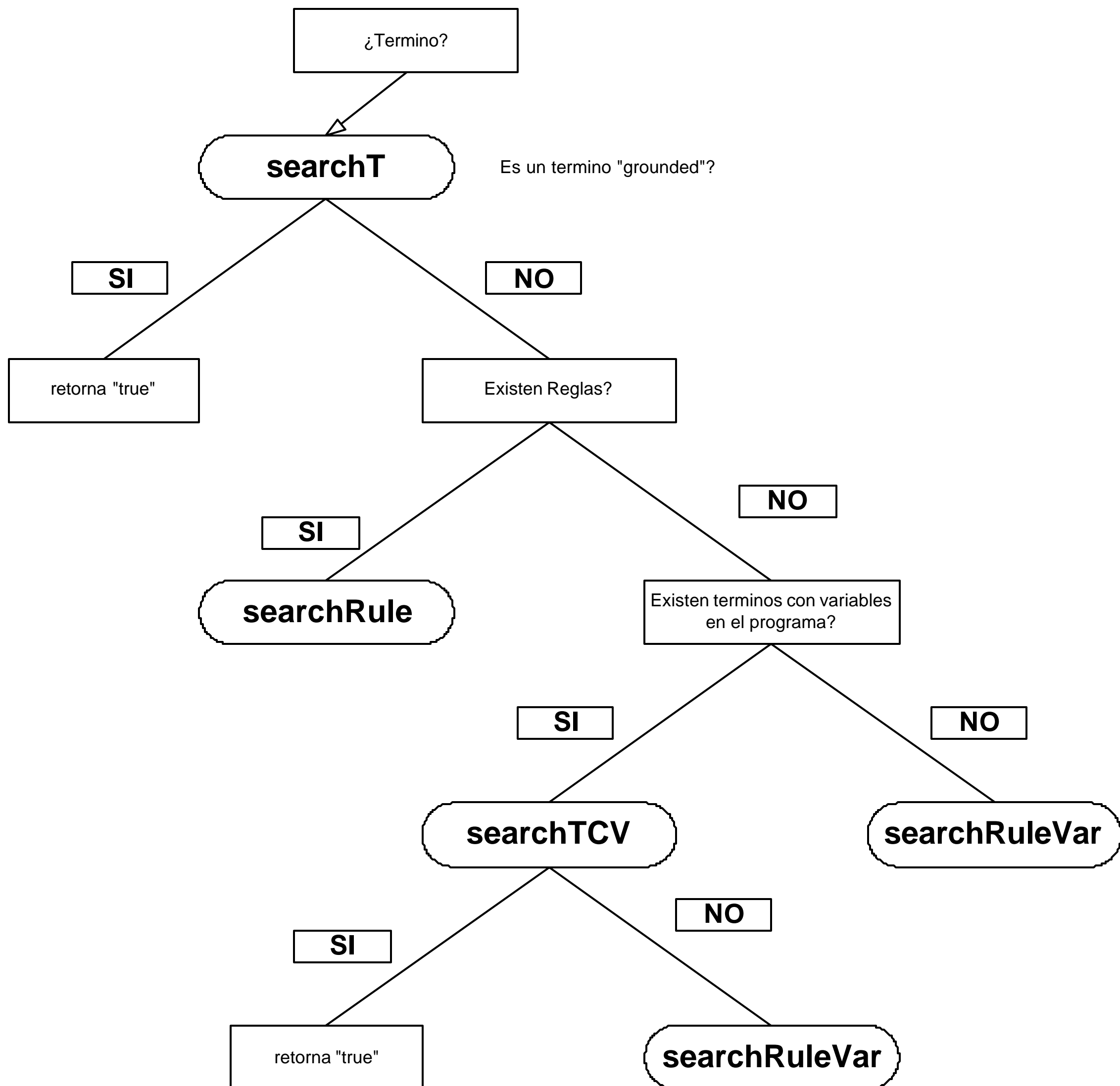

FIGURA 3.5. CONTROL DECLARATIVO SIN REGLAS

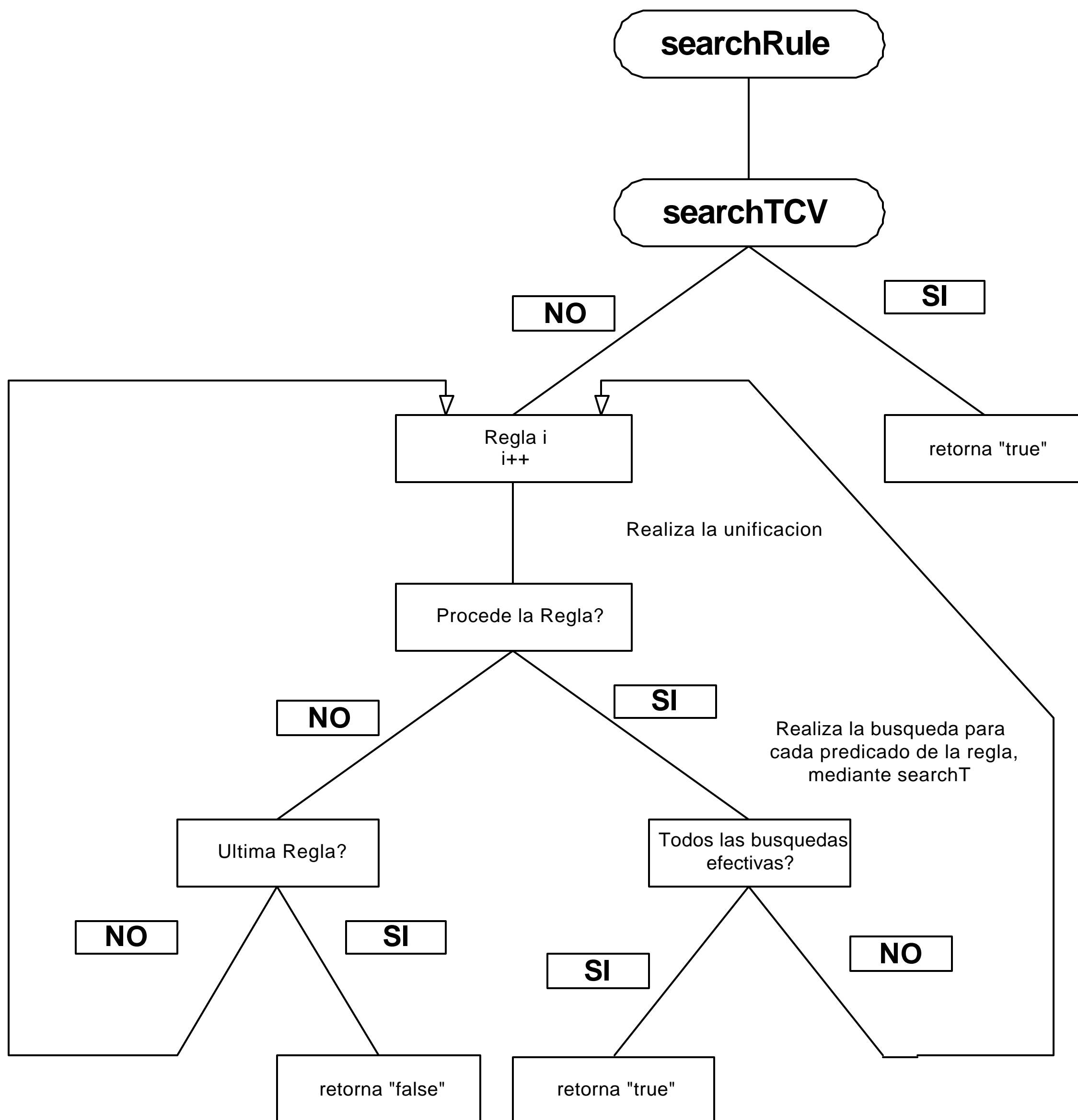

FIGURA 3.6. CONTROL DECLARATIVO CON REGLAS

En vista de que se deja que la máquina virtual de Java controle la recursividad, es necesaria la definición de un variable de control que permita la generación automática de las variables libres. Supónganse el caso de un programa en Prolog, similar al siguiente:

tiempo(1,1).
tiempo(F,1) :- M is F -1, tiempo(M,1).
actual(X): - tiempo(X,1). (3.9)

al realizar el seguimiento del programa en el espacio de búsqueda debido a una pregunta similar a:

?actual(3).

podemos observar que el flujo de información referente al llamado de reglas y generación de las preguntas o metas intermedias. En la figura siguiente se puede ver el mecanismo de seguimiento mediante un árbol que representa el espacio de búsqueda.

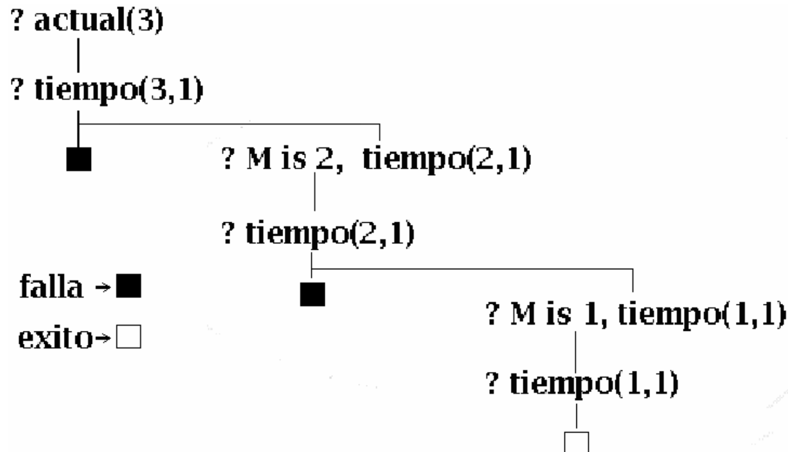

FIGURA 3.7. ÁRBOL DE BUSQUEDA DEL CONTROL DECLARATIVO

Para corroborar la formación del árbol de búsqueda a traves de los mecanismos del control declarativo, modificamos el código fuente de las clases: *Actual_1* y *Tiempo_2*, agregando un trazador para observar los métodos llamados en la consulta. El trazador tiene el formato de: *nombreClase.metodo - termino_1, …, termino_n, CBackT*. Al realizar la consulta *?actual(3)* se obtienen la siguiente traza:

Actual_1.searchT – 3, 1, 1
Actual_1.searchRule – 3, 1, 1
Tiempo_2.searchT – 3, 1, 1
Tiempo _2.searchRule – 3, 1, 1
Tiempo _2.searchT – 2, 1, 2
Tiempo _2.searchRule – 2, 1, 2
Tiempo _2.searchT – 1, 1, 2
// --------YES ----------
::: 3 (3.10)

Al realizar un análisis de los resultados obtenidos en (3.10), se puede observar que se genera un árbol de búsqueda similar al mostrado en la figura 3.7. Adicionalmente, se puede verificar la salida presentada en (3.10), a través de los diagramas de flujo que se muestran en las figuras 3.5 y 3.6, el orden de ejecución de los métodos *search** (*searchT, searchRule, searchRuleVar y searchTCV*) para el caso del programa (3.9) y la consulta planteada.

Existen dos puntos importantes que deben ser abordados como condiciones relevantes de la salida presentada en (3.10). Primero, se puede observar el proceso de generacion de consultas a la clase *Tiempo_2.java* de forma recursiva, y como es iniciada por el metodo *searchT*. Es decir, que la generacion de nuevas metas –de la forma *?tiempo(Hi, 1), en este caso -,* es lanzada unicamente por el metodo *searchT*, luego de lo cual, el control declarativo a traves del arbol de búsqueda sigue el flujo mostrado en las figuras 3.5 y 3.6.

En segundo lugar, como pudo observarse en (3.10) el último de los campos en cada uno de los metodos *search**, es una variable entera. Esta variable se denomina *CBackT,* y es definida en todos los metodos *search** del control declarativo y se utiliza para generar nuevas variables libres independientes para cada uno de los ciclos recursivos de búsqueda. Por ejemplo, si realizamos la consulta *?actual(9)* al programa (3.9) con los trazadores discutidos anteriormente. El resultado que se obtiene es el siguiente:

Actual_1.searchT – 9, 1, 1
Actual_1.searchRule – 9, 1, 1
Tiempo_2.searchT – 9, 1, 1
Tiempo _2.searchRule – 9, 1, 1
Tiempo _2.searchT – 8, 1, 2
Tiempo _2.searchRule – 8, 1, 2
Tiempo_2.searchT – 7, 1, 3
Tiempo _2.searchRule – 7, 1, 3
Tiempo _2.searchT – 6, 1, 4
Tiempo _2.searchRule – 6, 1, 4
Tiempo_2.searchT – 5, 1, 5
Tiempo _2.searchRule – 5, 1, 5
Tiempo _2.searchT – 4, 1, 6
Tiempo _2.searchRule – 4, 1, 6
Tiempo_2.searchT – 3, 1, 7
Tiempo _2.searchRule – 3, 1, 7
Tiempo _2.searchT – 2, 1, 8
Tiempo _2.searchRule – 2, 1, 8
Tiempo _2.searchT – 1, 1, 9
// --------YES ----------
::: 9 (3.11)

Puede observarse como la variable *CBackT,* va cambiando a media que descendemos en el árbol de búsqueda. Cada nueva hoja consultada en el árbol implica un aumento en el valor de la variable *CBackT* en una cantidad de 1. La variable CBackT, hará que los valores de las variables libres, sean inicializados de forma diferente para cada ciclo. El valor inicial de las variables esta dado por:

H1 = concatenar ("H1", CBackT)
H2 = concatenar (“H2", CBackT)
…
Hn = concatenar (“Hn", CBackT)

donde concatenar, es un método que permite variar el valor de inicialización de las variables según el valor CBackT. En nuestro caso no es consecutiva la numeración como el caso de Prolog, sino que depende del ciclo de búsqueda.

Para entender mejor la razón de ser de la variable *CBackT*, continuemos con el ejemplo (3.9) pero vamos a realizar una modificacion en el codigo de la clase Tiempo_2.java suprimiendo la variable *CBackT*, y adicionalmente modificamos los trazadores para incluir información de las variables libres. El formato de los trazadores utilizado fue *trazador_anterior: variables libres*. Realizamos luego la consulta *?actual(3)*.

```
Actual_1.searchT – 3, 1, 1:
Actual_1.searchRule – 3, 1, 1: H1
Tiempo_2.searchT – 3, 1, 1:
Tiempo _2.searchRule – 3, 1, 1: H1, H2
Tiempo _2.searchT – 2, 1, 1:
Tiempo _2.searchRule – 2, 1, 1: H1, H2
Tiempo _2.searchT – 1, 1, 1:
// --------YES ----------
::: 3/H1
```

Lo cual arroja un resultado incongruente. Veamos el mismo ejemplo pero ahora con la inclusión de la variable *CBackT.*

```
Actual_1.searchT – 3, 1, 1:
Actual_1.searchRule – 3, 1, 1: H1_1
Tiempo_2.searchT – 3, 1, 1:
Tiempo _2.searchRule – 3, 1, 1: H1_1, H2_1
Tiempo _2.searchT – 2, 1, 2:
Tiempo _2.searchRule – 2, 2, 1: H1_2, H2_2
Tiempo _2.searchT – 1, 1, 3:
// --------YES ----------
::: 3
```

Podemos concluir que dado que en cada llamada recursiva, dentro de un método search* conlleva a la inicialización de variables libres, de la forma *Hn = “Hn”, 0 –nombre Hn y aridad 0, y donde n es un entero-*; Si inicializamos las variables libre siempre con el mismo valor, -es decir sin el uso de *CBackT*- el procedimiento de encadenamiento hacia atrás perderá el rastro de las variables consultadas lo que origina respuestas incongruentes.

3.3.2. DEFINICION DE LAS CLASES DE CONVERSION A JAVA

Se diseñaron los mecanismos de traducción de tal forma, que sea relativamente sencilla la implementación de una mayor potencialidad y nuevas características posteriormente.

Llevar a cabo la conversión de Java a Prolog, requiere un análisis gramatical de las cláusulas del programa Prolog. Este proceso se conoce como *Parsing*, por su denominación en inglés. La denominación de análisis gramatical se fundamenta en el hecho que debe realizarse un "recorrido" por el programa en Prolog, en búsqueda de términos, chequeo de sintaxis de las cláusulas, entre otras tareas.

Una vez realizado el proceso de *parsing*, debe llevarse a cabo la generación de los archivos *.java* asociados al predicado en estudio, bajo el formato discutido en el apartado 3.3.1. Por lo cual, podemos determinar los dos procesos básicos para la conversión a clases: *parsing* y generación de clases.

En esta sección abordamos lo relacionado con el proceso de *parsing*. En la siguiente sección se tratará la información referente a la generación de las clases.

El programa de *parsing,* esta compuesto básicamente por tres clases denominadas *Parser_1, Parser_2* y *Parser_3*. Adicionalmente se requiere realizar llamadas a métodos de la clase *Operators*, debido a que en ésta residirán las operaciones válidas para la conversión. Por último, se definió una clase que almacenará el resultado del proceso de *parsing*.

La clase *Parser_1.java*, realiza la eliminación de líneas en blanco, tabulaciones, espacios en blanco, entre otros caracteres especiales. Adicionalmente chequea que cada una de las cláusulas termine con el símbolo terminal ´.´. En conjunción con la clase *Operators*, se etiquetan las operaciones con un símbolo adicional para su posterior conversión asociada a una instancia de la clase *Operators.* Generalmente la etiqueta consiste en un carácter o conjunto de caracteres que se anteponen al identificador de las operaciones.

La clase *Parser_2.java*, chequea carácter a carácter, para descomponer cada una de las cláusulas en términos atómicos, almacenándolos en una estructura de similar a,

```
nombre = nombre_del_Termino;
aridad = aridad_asociada_al_Termino;
subTerminos = Vector Termino;
```

cuya estructura corresponde a la clase definida como *ParseTerm*. En el anexo A se muestran la totalidad de métodos implementados en esta clase.

Cada una de las cláusulas es analizada utilizando la clase *StringTokenizer*, propia de la API de Java. La clase *StringTokenizer* permite convertir una cadena de caracteres en una secuencia de *tokens* o fragmentos de substrings. Los *tokens* corresponden a palabras completas con las cuales podemos construir los términos *ParseTerm*.

La clase *Parser_3.java*, almacena los átomos con formato estipulado en la clase *ParseTerm*, en dos vectores: *body* y *head*. El almacenamiento tiene como condición el ordenamiento de los átomos. Para aclarar este proceso, supóngase el programa:

nuevo(s,d).
viejo(k,l):- algo.
nuevo(A,F):- viejo(A,F), not(A=F). (3.12)

Luego del proceso de *parsing*, este programa daría origen a los siguientes vectores:

Head	body
nuevo(s,d)	Null
nuevo(A,F)	viejo(A,F) - not(A=F)
viejo(k,l)	algo

Puede verse el proceso de ordenamiento de las cláusulas del programa (3.12). Es importante resaltar que el contenido de los vectores *head* y *body*, son términos con el formato de la clase *ParseTerm.*

El resultado del proceso de *parsing*, representado por los dos vectores generados por la clase *Parser_3*, constituyen los parámetros de entrada a la clase de generación de clases denominada *ScriptW*. En la figura 3.8, se muestra el diagrama UML de las clases asociadas al proceso de *parsing*.

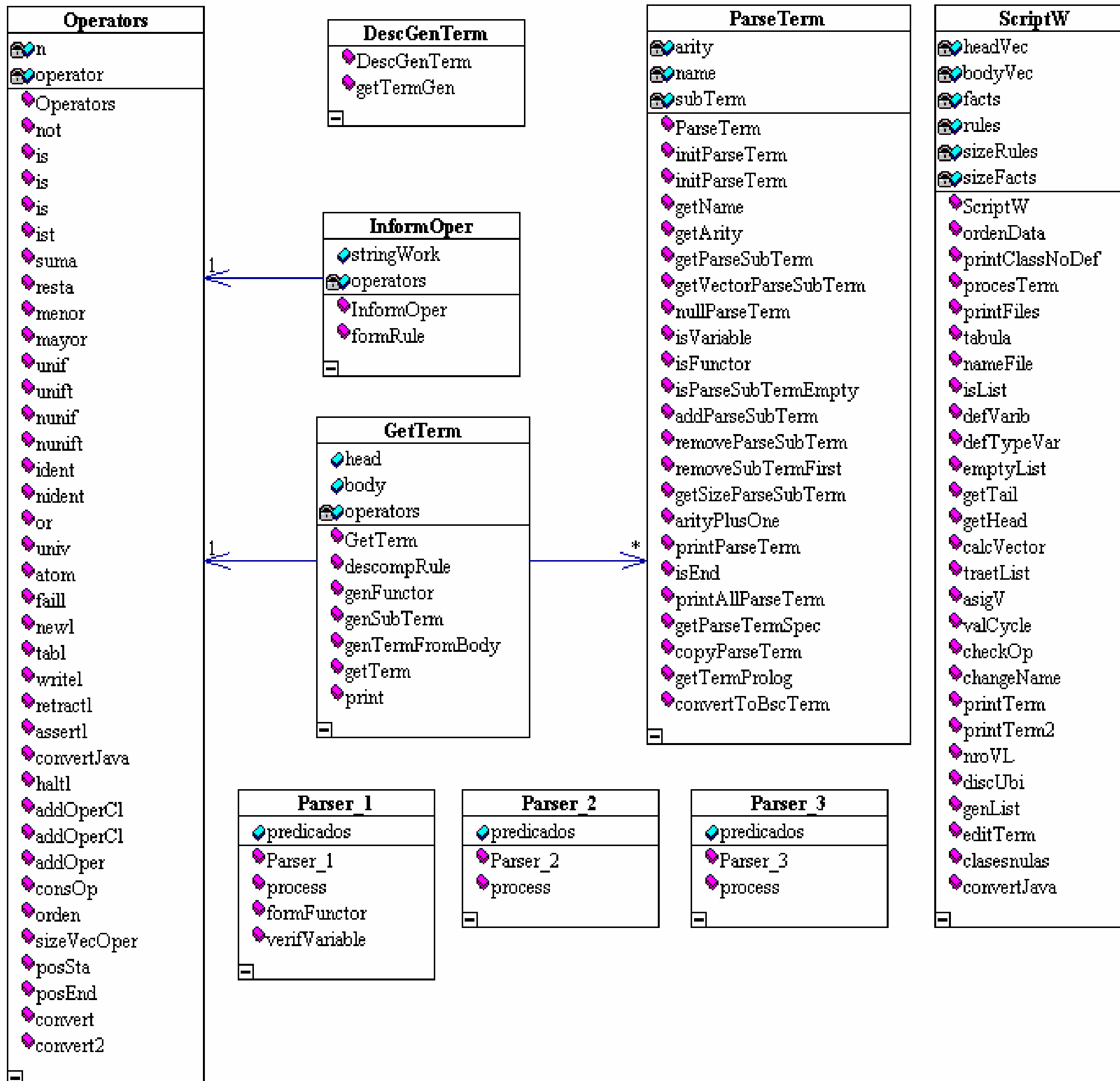

FIGURA 3.8. CLASES ASOCIADAS AL PROCESO DE PARSING

En la última versión de nuestro programa, las clases de *parsing* se integraron a las clases *GenClass, GenQ, GenIC* y *GenDef*.

3.3.3. GENERACION DE LAS CLASES JAVA

De la figura (3.15) se desprende la existencia de una clase denominada *ScriptW*. En nuestra propuesta se definió la clase *ScriptW* que toma como argumentos de entrada, los vectores provenientes de la clase *Parser_3*. Esta clase permite generar los archivos *.java* en las cuales se incluyan cada una de las estructuras discutidas en la sección 3.3. Para

cada uno de los átomos existentes en el vector *head*, -sin multiplicaciones- se generar un archivo con el nombre:

NombrePredicado_aridadAsociada.java

Se utiliza el nombre del predicado, seguido por el símbolo de "underscore", más la aridad correspondiente a dicho predicado.

La clase *ScriptW* define un método para procesar los datos provenientes de los vectores *head* y *body*. El método se denomina:

public void ordenData(Vector, Vector)

Desde el método *ordenData* se procede a llamar al método principal de impresión:

public Vector printFiles(Vector, Vector, Vector, int, int, int, String)

Es importante resaltar que el método *printFiles* devuelve un vector, en el cual residen los nombres de todos los predicados asociados al programa en Prolog.

El vector resultante de la operación de *printFiles*, se introduce como parámetro al método:

Vector procesTerm(Vector, Vector)

que arroja como resultado, el compendio de todos los predicados que no han sido definidos con un átomo o con una regla en cuya cabecera exista una definición de este predicado. Por lo cual, debe generarse una clase asociada a cada uno de los predicados, mediante el siguiente método:

void printClassNoDef(Vector)

Está clase especial en Java, la denominamos "*clase no definida*". La estructura de la clase no definida, es la siguiente,

```
class Predicado_aridad{
        //atributos
        ListLinked facts = new ListLinked();
        int numberRules;
        int numberAtoms;
        Term termA1;            //termino_1
        ...
        Term termAn;            //termino_n
        //métodos básicos
        public int getArity () {...}
        public int getNumberTerms() {...}

        //metodos principales de control
        public boolean searchT(Term, Term, ..., Term, int) {...}
}
```

El método *searchT* de las *clases no definidas*, siempre devuelve un valor de asertividad igual a *false*. La importancia de la creación de *clases no definidas*, se puede ver mejor con un ejemplo. Supóngase el siguiente programa en Prolog y sobre el cual se realiza la consulta *?padre(m,h),*

```
padre(a,d).
padre(X,G) :- hijo(X,H), hijo(G,H).
```

La respuesta en un IDE de Prolog, es "no". Es decir, con el conjunto de reglas existentes en el programa, no puede derivar la consulta *?padre(m,h).* Sin la existencia de *clases no definidas* y dada la estructura de generación de clases discutidas previamente, involucraría la generación de un error por parte de la JVM, ya no podría encontrar la definición de la clase *Hijo_2*. Es decir, en el caso de Java se generará una excepción *classNotFound* y en Prolog una falla debido a que predicado no está definido. Para mantener la misma semántica de Prolog, los métodos de las "clases no definidas" de nuestra propuesta siempre "fallan", es decir, cualquier consulta hecha a esos métodos retornarán un valor de falsedad.

Adicionalmente se define varios métodos que permiten manipular los términos del tipo *ParseTerm*, para formar las estructuras de la sección 3.3.1. En la figura 3.9 se muestra la información referente al diagrama UML de la clase *ScriptW*.

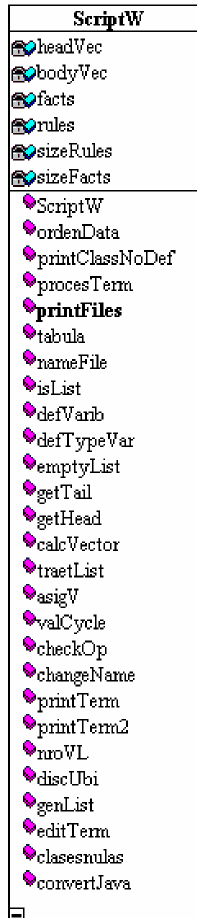

FIGURA 3.9. DIAGRAMA DE LA CLASE SCRIPTW

Definidas las clases de *parsing* y la generación de clases java asociadas a los predicados de Prolog. Se generaron dos programa, uno en modo gráfico y otro en modo texto.

El programa gráfico, consiste en una interfaz que utiliza las clases AWT de Java. La GUI correspondiente permite introducir el nombre del archivo, donde se encuentra el programa en Prolog que se desea convertir a sus clases equivalentes en Java. En el programa se instancian las clases descritas en la figura 3.8. El nombre del programa es:

GenerarClases

El diagrama UML del generador de clases en modos gráfico se puede observar en la figura 3.10,

FIGURA 3.10. PROGRAMA GENERADOR DE CLASES

El programa en modo texto se denomina *GenClass*, y posee las mismas características del programa en modo grafico. Su invocación se lleva a cabo de la siguiente forma:

%java GenClass nombre_del_archivo_Prolog

Para comprobar el funcionamiento de los programas generadores de clases, supóngase que se diseña el siguiente programa en Prolog:

```
padre(C,V).
padre(X,Y) :- hijo(X,Z), hijo(Y,Z), not(X=Y).          (3.13)
```

Se puede ejecutar el programa en cualquiera de las dos versiones, modo gráfico o texto. A continuación, se muestra el resultado, luego de aplicar el programa *GenerarClases* a un archivo con las dos cláusulas del programa (3.13)

```
import progtojav.*;
import java.util.*;

/**
* <p>Definicion de la clase asociada al predicado indicado por el nombre de la clase y la aridad</p>
* @author Jhon Edgar Amaya
* @version 1.2
*/

class Padre_2{

	/**
	* Atributos de la clase
	*/

	 int numberAtoms;
	 int numberRules;
	 int arity;

	 Term termA1;
	 Term termA2;
	 LinkedList facts = new LinkedList();

	 /**
	 * Definicion del contructor de la clase
	 */

	 Padre_2() {
		numberAtoms = 1;;
		numberRules = 1;
		arity = 2;
		termA1 = new Term();
		termA2 = new Term();
	 }

	 /**
	 * <p>Title: addElement </p>
	 * <p>Description: Permite incluir hechos a la clase relacionada con el predicado</p>
	 * <p>los terminos deben ser de tipo constante o listas </p>
	 * @param  Term, ... ,Term
	 */

	public boolean addElement(Term X1, Term X2){
		if ( X1.isAtom() && X2.isAtom() ){
			facts.add(X1);
			facts.add(X2);
			return true;
```

```
        } else
                return false;
}

/**
 * <p>Title: removeElement </p>
 * <p>Description:Permite eliminar hechos la clase relacionada con el predicado</p>
 * <p>los terminos deben ser de tipo constante o listas </p>
 * @param  Term, ... ,Term
 */

public boolean removeElement(Term X1, Term X2){
        if ( X1.isAtom() && X2.isAtom() ){
                facts.remove(X1);
                facts.remove(X2);
                return true;
        } else
                return false;
}

/**
 * <p>Title: getArity </p>
 * <p>Description: Devuelve la aridad del predicado </p>
 * @param void
 * @return int
 */

public int getArity(){
        return arity;
}

/**
 * <p>Title: getNumberRules </p>
 * <p>Description: Devuelve el numero de reglas asociadas al predicado </p>
 * @param void
 * @return int
 */

public int getNumberRules(){
        return numberRules;
}

/**
 * <p>Title: getNumberFacts </p>
 * <p>Description: Devuelve el numero de atomos asociadas al predicado </p>
 * @param void
 * @return int
 */

public int getNumberFacts(){
        return numberAtoms;
}

/**
```

```
 * <p>Title: obtenerAtomos </p>
 * <p>Description: Devuelve la lista de terminos </p>
 * @param void
 * @return LinkedList
 */

public LinkedList obtenerAtomos(){
          return facts;
 }

 /**
 * <p>Title: getNumberElements </p>
 * <p>Description: Devuelve la cantidad de terminos almacenados en la lista </p>
 * <p>de hechos de la clase </p>
 * @param void
 * @return int
 */

public int getNumberElements(){
          return facts.size();
 }

 /**
 * <p>Title: getNumberTerms </p>
 * <p>Description:Retorna la cantidad de term. almacenados en la lista/aridad </p>
 * @param void
 * @return int
 */

public int getNumberTerms(){
          return facts.size()/arity;
 }

 /**
 * <p>Title: getTerm </p>
 * <p>Description: Devuelve el i-esimo termino en la lista de terminos </p>
 * @param int
 * @return Term
 */

public Term getTerm(int i){
          return ((Term) facts.get(i));
 }

 /**
 * <p>Title: printAnswer </p>
 * <p>Description: Devuelve el resultado de la busqueda </p>
 * @param  Term  Term
 * @return String
 */

public String printAnswer(Term termIn, Term termExt){
          String retorno = "";
```

```
		if (termIn.equals(termExt)){
			retorno = "::: "+termIn.printListTerm();
		} else {
			retorno = "::: "+termExt.printListTerm() + "/" + termIn.printListTerm();
		}
		return retorno;
	}

	/**
	* <p>Title: calcGr </p>
	* <p>Description:Calcula la cantidad de cifrassignificativas del enterodecontrol</p>
	* @param  int
	* @return int
	*/

	public int calcGr(int i){
		int retorno = 1;

		if (i<10)		{ retorno = 10;
		} else if (i < 100) {	retorno = 100;
		} else if (i < 1000) { retorno = 1000;
		} else { retorno = 10000;
		}
		return retorno;
	}

	/**
	* <p>Title: conVarLi </p>
	* <p>Description: Calcula la variable libre en proceso</p>
	* @param  int int int
	* @return int
	*/

	public int conVarLi(int InC, int DigC, int VarC){
		return ((DigC * VarC) + InC);
	}

	/**
	* <p>Title: searchT </p>
	* <p>Description: Chequea la existencia de hechos en la clase </p>
	* <p>si existe el hecho devuelve la condicion de verdad, </p>
	* <p>en caso contrario cheque si hay reglas, si no hay devuelve </p>
	* <p>la condicion de falso </p>
	* @param  Term, ... ,Term int
	* @return boolean
	*/

	public boolean searchT(Term X1, Term X2, int CBackT){
		int i = facts.size();
		boolean retorno = false;
		CheckChain condf = new CheckChain();

		if ((i != 0) && condf.checkTermGround(X1) && condf.checkTermGround(X2) ){
			while(i > 0) {
```

```
			if ((X1.equals((Term)facts.get(i-2))) && (X2.equals((Term)facts.get(i-1)))){
					termA1.sustTerm((Term) facts.get(i-2));
					termA2.sustTerm((Term) facts.get(i-1));
					retorno = true;
					break;
				}
				i = i - arity;
			} //fin del while
		} //fin del if

		if (retorno==false){
			if (numberRules != 0){
				retorno = searchRule(X1, X2, CBackT);
			} else {
				retorno = searchTCV(X1, X2,CBackT);
				if(! retorno){
					retorno = searchRuleVar(X1, X2);
				}
			}
		} //fin del if
		return retorno;
	}

	/**
	* <p>Title: searchWOR </p>
	* <p>Description: Chequea la existencia de hechos en la clase </p>
	* <p>si existe el hecho devuelve la condicion de verdad, </p>
	* <p>en caso contrario devuelve la condicion de falso </p>
	* @param  Term, ... ,Term
	* @return boolean
	*/

	public boolean searchWOR(Term X1, Term X2){
		int i = facts.size();
		boolean retorno = false;
		CheckChain condf = new CheckChain();

		if ((i != 0) && condf.checkTermGround(X1) && condf.checkTermGround(X2) ){
			while(i > 0) {
			if((X1.equals((Term)facts.get(i-2))) && (X2.equals((Term) facts.get(i-1)))){
					termA1.sustTerm((Term) facts.get(i-2));
					termA2.sustTerm((Term) facts.get(i-1));
					retorno = true;
					break;
				}
				i = i - arity;
			} //fin del while
		} //fin del if

		return retorno;
	}

	/**
	* <p>Title: getAtomWithVar </p>
```

```
 * <p>Description: Devuelve un atomo a partir de una solicitud de una variable </p>
 * @param  Term, ... ,Term
 * @return boolean
 */

public Vector getAtomWithVar(Term X1, Term X2){
        int i=0;
        int j=facts.size();
        Vector retorno=new Vector();
         if (X2.isVariable())  { i = 1;}
        switch(i){
        case 0:
                while(j>0){
                        if (facts.get(j-1)==X2){
                                retorno.addElement((String) facts.get(j-2));
                        }
                        j=j-arity;
                }
        break;
        case 1:
                while(j>0){
                        if (facts.get(j-2)==X1){
                                retorno.addElement((String) facts.get(j-1));
                        }
                        j=j-arity;
                }
        break;
         }//fin del switch
        return retorno;
}

 /**
 * <p>Title: searchRuleVar </p>
 * <p>Description: Permite la busqueda de variables en una regla </p>
 * @param  Term, ... ,Term
 * @return boolean
 */

public boolean searchRuleVar(Term X1, Term X2){
        boolean retorno = false;
        int i = 0;
        Vector argum = new  Vector();
        Vector mGUnf = new  Vector();
        Vector varbl = new  Vector();
        UnifTerm eleUnig = new UnifTerm();
        Term H1 = new Term("H1",0);
        Term H2 = new Term("H2",0);
        while(i < facts.size()){
                H1 = (Term) facts.get(i+0);
                H2 = (Term) facts.get(i+1);
                argum = eleUnig.formVector(X1, X2);
                varbl = eleUnig.formVector(H1, H2);
                mGUnf = eleUnig.unify(argum, varbl, arity);
                if (! eleUnig.getCondUnify()){
```

```
                    termA1.sustTerm(H1);
                    termA2.sustTerm(H2);
                    retorno = true;
                    break;
                } //fin del if
                i += arity;
            } //fin del while
            return retorno;
        }

        /**
         * <p>Title: searchRule </p>
         * <p>Description: Chequea las reglas asociadas con el predicado</p>
         * @param  Term, ... ,Term int
         * @return boolean
         */

        public boolean searchRule(Term X1, Term X2, int CBackT){

            Term H1 = new Term("H1"+CBackT,0);
            Term H2 = new Term("H2"+CBackT,0);
            Term H3 = new Term("H3"+CBackT,0);
            Term T1 = new Term();
            Term T2 = new Term();
            Vector argum = new  Vector();
            Vector argUf = new  Vector();
            Vector mGUnf = new  Vector();
            Vector varbl = new  Vector();
            UnifTerm eleUnig = new UnifTerm();
            int contBack = calcGr(CBackT);
            boolean retorno = false;


            for(int j=0; j < getNumberTerms(); j++){
                H1.changeTerm("H1"+CBackT,0);
                H2.changeTerm("H2"+CBackT,0);
                argum = eleUnig.formVector(X1,X2);
                varbl = eleUnig.formVector(H1,H2);
                mGUnf = eleUnig.unify(argum, varbl, arity);
                varbl = eleUnig.buildSusVV(varbl,mGUnf);
                H1 = (Term) varbl.elementAt(0);
                H2 = (Term) varbl.elementAt(1);

                if((! H1.isConstant()) | | (! H2.isConstant())){
                    if(H1.isVariable()){H1.changeTerm((Term) facts.get(j*arity+0));}
                    if(H2.isVariable()){H2.changeTerm((Term) facts.get(j*arity+1));}
                    if(searchWOR(H1, H2)){
                        retorno = true;
                        termA1 = H1;
                        termA2 = H2;
                        return retorno;
                    }
                }
```

```
}

Hijo_2 hijo_2i = new Hijo_2();
//########################################################
//######### Regla # 1 ########
//########################################################
H1.changeTerm("H1"+CBackT,0);
H2.changeTerm("H2"+CBackT,0);
H3.changeTerm("H3"+CBackT,0);
argum = eleUnig.formVector(X1,X2);
//Info de las T
T1.asigValue(H1);
T2.asigValue(H2);
argUf = eleUnig.formVector(T1,T2);
mGUnf = eleUnig.unify(argum, argUf, arity);
if (! eleUnig.getCondUnify()){
varbl = eleUnig.formVector(H1,H2,H3);
varbl = eleUnig.buildSusVV(varbl,mGUnf);
H1 = (Term) varbl.elementAt(0);
H2 = (Term) varbl.elementAt(1);
H3 = (Term) varbl.elementAt(2);
int i1=0;
while(i1==0 || i1 < hijo_2i.getNumberTerms()){
        if (hijo_2i.getNumberTerms()!= 0){
                H1 = eleUnig.unify2term ( (Term) varbl.elementAt(0), (Term)
                hijo_2i.getTerm( i1* (hijo_2i.getArity()) +0));
                H3 = eleUnig.unify2term( (Term) varbl.elementAt(2), (Term)
                hijo_2i.getTerm( i1* (hijo_2i.getArity()) +1));
                H2 = eleUnig.unify2term( (Term) varbl.elementAt(1), (Term)
                hijo_2i.getTerm(i1* (hijo_2i.getArity()) +0));
        }

        if (hijo_2i.searchT(H1,H3,CBackT)== true){
                H1.asigValue(hijo_2i.termA1);
                H3.asigValue(hijo_2i.termA2);
                if (hijo_2i.searchT(H2,H3,CBackT)== true){
                        H2.asigValue(hijo_2i.termA1);
                        H3.asigValue(hijo_2i.termA2);
                        if ( ! (oper.unif(H1, new Term("Y", 0))== true) ){
                                retorno = true;
                                varbl = eleUnig.formVectorSpec( conVarLi(
                                CBackT, 1, contBack), H1, conVarLi( CBackT, 2,
                                contBack), H2, conVarLi( CBackT, 3, contBack),
                                H3,conVarLi(CBackT,4,contBack),H4);

                                termA1 = eleUnig.buildSusTV(T1, varbl);
                                termA2 = eleUnig.buildSusTV(T2, varbl);
                                return retorno;
                        }
                }
        }

        i1++;
        } //fin de ciclo i1
```

```
            }// fin del if
            return retorno;
    } // fin de metodo

    /**
    * <p>Title: searchTCV </p>
    * <p>Description: Chequea la existencia de hechos con variables en la clase </p>
    * <p>si puede realizar la unificacion devuelve la condicion de verdad, </p>
    * <p>en caso contrario devuelve la condicion de falso </p>
    * @param  Term, ... ,Term int
    * @return boolean
    */

    public boolean searchTCV(Term X1, Term X2, int CBackT){
            boolean retorno = false;
            Term H1 = new Term("H1"+CBackT,0);
            Term H2 = new Term("H2"+CBackT,0);
            Term H3 = new Term("H3"+CBackT,0);
            Term H4 = new Term("H4"+CBackT,0);
            Term T1 = new Term();
            Term T2 = new Term();
            Vector argum = new  Vector();
            Vector argUf = new  Vector();
            Vector mGUnf = new  Vector();
            Vector varbl = new  Vector();
            UnifTerm eleUnig = new UnifTerm();
            int contBack = calcGr(CBackT);
            //#########################################################
            // Termino con variables # 1
            //#########################################################
            T1.asigValue(H1);
            T2.asigValue(H2);

            argum = eleUnig.formVector(X1,X2);
            argUf = eleUnig.formVector(T1,T2);
            mGUnf = eleUnig.unify(argum, argUf, arity);
            if (! eleUnig.getCondUnify()){
                    retorno = true;
                    termA1 = eleUnig.buildSusTV(T1, mGUnf);
                    termA2 = eleUnig.buildSusTV(T2, mGUnf);
                    return retorno;
            }

            return retorno;
    }
}
```

FIGURA 3.11. CODIGO DE UNA CLASE CREADA POR GENERADOR CLASES

Para aclarar un poco mejor la estructura de la clase se presenta el diagrama UML asociado a esta clase generada del programa (3.13),

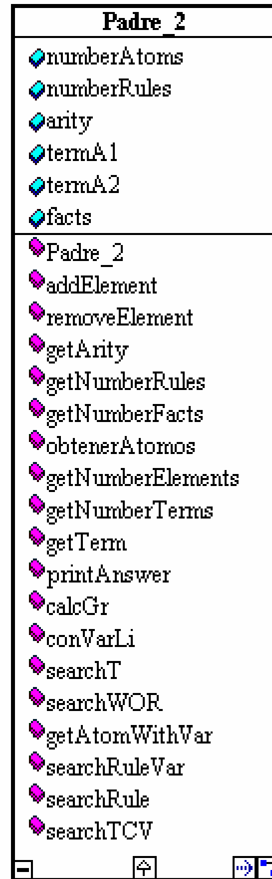

FIGURA 3.12. DIAGRAMA UML DE UNA CLASE CREADA POR GENERADOR CLASES

3.3.4. DEFINICIÓN DE LAS CLASES FUNCIONALES

La implementación de mecanismos similares al funcionamiento de los lenguajes declarativos, hace necesario la definición de cuatro clases funcionales básicas. Se utilizó la denominación de clases funcionales, debido a que éstas cumplen la *función* de definición y manipulación de términos, en todos aquellos programas que se desarrollen bajo nuestra propuesta. En estas clases funcionales se incluyen la definición de operadores, proceso de resolución, definición de términos, entre otros. En las siguientes cuatro secciones se discutirán las clases funcionales asociadas a la propuesta. Las clases funciones son: *Term, BscTerm, UnifTerm* y *Operators*.

3.3.4.1. CLASE TERM

La clase *Term* representa el núcleo de todos los procesos de nuestra implementación, ya que todos los elementos propios de los lenguajes declarativos radican en la manipulación de predicados y consecuentemente en términos.

Cada término esta constituido por un conjunto de términos básicos, agrupados en un estructura tipo vector. El término puede ser una lista o un término individual. El término básico, es definido por un término en la extensión clásica de la definición, tal y como se muestra en la figura 2.4. En el software, se implementa esta clase con el nombre de *BscTerm*, y se discutirá con más detalle en sección siguiente.

Se puede observar en la figura 3.13, como el único atributo de la clase es una variable de nombre *termList*, que es de tipo Vector y en la cual se almacenan los términos básicos. Esta definición en forma de vector permite almacenar la información de los términos como una "colección de datos", y representar la estructura de datos manejada por Prolog conocida como lista de términos. Es importante mencionar que Java no maneja el concepto de punteros, tal como lo hace lenguaje C, o como se estructuran los registros en la WAM, para localizar los términos y modificar el valor de éstos [AIK-1]. Resulta conveniente una estructura tipo vector para el manejo de las listas de términos, ya podemos hacer facil uso de los métodos para la manipulación de vectores que presenta la API de Java, adicionalmente, mantenemos la misma sintaxis de trabajo como si se tratara una pseudopila en Prolog. En resumen, la idea detrás de la estructura de *Term,* es formar una lista de términos y manejarlo de forma transparente.

FIGURA 3.13. DIAGRAMA UML DE LA CLASE TERM

La forma de invocar los términos en nuestra propuesta se lleva cabo de la siguiente forma, supóngase los términos:

10 *se iniciaría como:* *Term("10", 0)*

hijo(hola, U) *se iniciaría como:* *Term("hijo", 2, "hola", 0, "U", 0)*

Se definen adicionalmente en la clase *Term* todo lo referente a la manipulación de los términos, incluyendo los cambios de nombre, obtención de aridad y el nombre correspondiente, entre otros métodos. En el anexo A se presentan cada uno de los métodos definidos en la clase *Term*.

3.3.4.2. CLASE BSCTERM

Son considerados los términos básicos con los cuales se estructura todo término manejado por nuestra propuesta. La clase *BscTerm* está estructurada como un objeto cuyos atributos son similares a *ParseTerm*, es decir,

nombre = nombre_del_Termino;
aridad = aridad_asociada;
subTerminos = Vector Termino;

Para entender la representación de los términos básicos, vemos unos ejemplos.

hola. *Se representaría como BscTerm("hola", 0 , null).*

f(V) . *Se representaría como BscTerm("f",1 , BscTerm("V", 0 , null)).*

f(V, C) . *Se representaría como BscTerm("f",2,{BscTerm("V",0,null),BscTerm("C",0 , null)}).*

De los ejemplo descritos se puede concluir que el objeto *BscTerm*, se representa como,

BscTerm(String, int , (BscTerm(...) , ... ,BscTerm(...))). *(3.14)*

De (3.14) se desprende además, la necesidad de definir varios constructores para la clase. La idea subyacente es abarcar la construcción de términos que posean diferentes aridades.

Se definieron los métodos para la inicialización de los términos básicos, consulta y modificación del nombre del término, aridad y subtérminos, entre otros; para la manipulación de los términos básicos.

Se incluyen en la clase métodos referentes para saber si un determinado término básico es una variable, una constante, una lista o un símbolo de *no importa*. Adicionalmente se definen métodos para la comparación de términos básicos, por ejemplo:

public boolean equal(BscTerm k)

public boolean equal(String name)

Para facilitar la depuración de términos básicos, se desarrollaron métodos que permiten visualizar el contenido del objeto término básico, como es el caso de,

public String printBscTerm()

En el anexo A se presentan cada uno de los métodos definidos en la clase *BscTerm*. La representación de *BscTerm* en el lenguaje UML se muestra en la figura 3.14.

FIGURA 3.14. DIAGRAMA UML DE LA CLASE BSCTERM.

3.3.4.3. CLASE UNIFTERM

El proceso de resolución se lleva a cabo mediante la definición de un algoritmo, que se incluye como parte del método *searchRule,* en cada una de las clases asociadas a los predicados. La resolución utiliza un algoritmo basado en SLD [HOG-1].

Existen varios procedimientos de prueba del mecanismo de la resolución, que determinan cuales deben ser las estrategias de búsqueda para lograr derivar una cláusula vacía justo como en los compiladores Prolog, y por lo tanto, sea considerado que se ha hallado la respuesta a una consulta. Esta cláusula vacía se logra formulando una pregunta, en este caso, una o varias cláusulas negativas o queries, que al aplicarle las reglas de inferencia con los predicados y reglas existentes en la base de conocimiento o programa de Prolog, se inicia el procedimiento de prueba para intentar obtener una cláusula vacía.

La clase *UnifTerm* tiene tres conjuntos básicos de metodos. El primero, consiste de los metodos encargados de calcular el unificador más general de un par de términos, a saber,

public Vector unify (Vector , Vector , int)
public Vector unify (Vector , Vector)

Los métodos *unify* se encargan de realizar el proceso de unificación, por medio de la implementacion del algoritmo de Robinson, tal como se mostró en la seccion 2.8.4.

El segundo grupo es el encargado de realizar el proceso de sustitución. Se diseñaron tres métodos para tales fines:

public Term buildSusTV (Term, Vector)
public Vector buildSusVV (Vector, Vector)
public BscTerm buildSusBTT (BscTerm, Term, Term)

El último grupo corresponde a los métodos que permiten formar los vectores, que llevan a cabo la manipulación del unificador más general. El formato de estos métodos es:

public Vector formVector(Term, Term, …, Term)

La máxima cantidad de términos soportados por los métodos de *formVector* es de 12. Lo anterior implica que se puede llevar la unificación entre términos con una aridad menor a 13.

En la figura 3.15, se muestra el diagrama UML de la clase de UnifTerm y su relación con las clases Term y BscTerm.

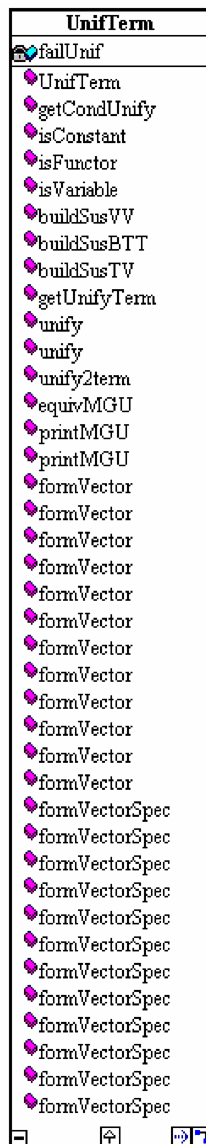

FIGURA 3.15. DIAGRAMA UML DE LA CLASE UNIFTERM

3.3.4.4. CLASE OPERATORS

Para agregar mayor funcionalidad a un programa que utilice una forma declarativa, se requiere la definición de las operaciones básicas, tales como las operaciones aritmeticas. En algunos compiladores a este tipo de funciones básicas, recibe el nombre *Built-in* ó constructores [SPR-1][JPR-1]. Para resolver consultas especiales se requiere que este tipo constructores sean definidos, inclusive en el lenguaje Java existe una paquete especial para la manipulación matemática java.math.*.

Se procedió a definir una clase donde se agrupan las operaciones aritméticas básicas de Prolog, como por ejemplo, la suma, la asignación numérica (*is* su nombre en ingles), entre otras. Se pretende brindar la suficiente flexibilidad para que sean incluidas nuevas funcionalidades como se discute a continuación.

En Prolog los operadores se definen de la siguiente manera:

> *:-op(precedencia, tipo, nombre)*
>
> *Precedencia es el valor de asociado con la definición de precedencia de las operaciones, entre mayor sea el valor de precedencia más prontamente debe realizarse la operación.*
>
> *Tipo, indica el tipo de operación que será llevada a cabo, bien sea infija, postfija o prefija*
>
> *Nombre indica el símbolo o conjunto de símbolos que representan la operación.*

Por ejemplo, la definición de las operaciones en Prolog puede verse en la figura 3.16 [SBP-1]. Se incluye el valor de precedencia, asi como los símbolos asociados para la representación de la operación, incluyendo el formato de ejecución, bien sea infija, postfija o prefija.

```
:- op(  1100,  xfy,   ; ).
:- op(  1050,  xfy,   -> ).
:- op(  1000,  xfy,   ',' ).
:- op(   900,  fy,    not ).
:- op(   700,  xfx,   =, is, =.., ==, \==,  =:=, =\=, <, >, =<, >=, ?=, \= ).
:- op(   661,  xfy,   `.' ).
:- op(   500,  yfx,   +, -, /\, \/ ).
:- op(   500,  fx,    +, - ).
:- op(   400,  yfx,   *, /, //, <<, >> ).
:- op(   300,  xfx,   mod ).
:- op(   200,  xfy,   ^ ).
```

FIGURA 3.16. DEFINICION DE OPERADORES EN PROLOG

Para entender la importancia de los valores de precedencia, supóngase un par de términos de la forma:

a(X is 2), not b,

si utilizamos como referencia el valor de precedencia mostrada en la figura 3.16, el compilador ejecutaría las instrucciones de la siguiente forma:

Oper(,)(a(oper(is)(X,2), oper(not)(b)).

Puede observarse que la primera operación ejecutada es aquella que tenga un mayor valor de precedencia. Estas premisas que describen el orden de precedencia de las operaciones a ejecutar, deben incorporarse en este esquema para que el programador pueda definir el orden de ejecución, así como la inclusión de nuevas operaciones.

Adicionalmente, se pueden ver que la notación infija debe convertirse a un formato del tipo prefijo para un manejo más dúctil de las operaciones. Se observa que el formato infijo se representa como *xfy*. En nuestro caso utilizamos la notación prefija de tal forma de implementar las operaciones como métodos propios de la clase *Operators*. La estructura de los métodos de la clase, supone que las operaciones son declaradas de forma prefija, además fue diseñada para permitir la sobrecarga de métodos.

La clase *Operators* consta de 2 atributos, a saber: *n y operator*. El primero de ellos, es decir *n*, es una variable entera que indica la cantidad de operaciones existente en momento particular. El atributo *operator* es un vector de objetos tipo *Signs.*

La clase *Signs*, posee tres atributos básicos: *codeProlog, signHere* y *valuePre.* El atributo *codeProlog* es el campo que contiene el conjunto de símbolos asociados a la representación de una operación particular, por ejemplo "+", "is", etc. El atributo *signHere* es la representación en la clase *Signs* de la operación en cuestión, es decir, supóngase que un programa contiene el operador asociado a la suma "+", entonces los métodos de *convert y convert2* se encargarán de ubicar la operación definida en el atributo *codeProlog* y los sustituirán por la representación que se encuentra a su vez en el atributo *signHere*.

El campo *valuePre* está asociado con el valor de precedencia de forma similar a Prolog, de hecho la sustitución realizada por los métodos *convert y convert2,* toma en cuenta la precedencia para realizar el proceso de sustitución.

La mayoria de las operaciones básicas tienen al menos dos métodos que los definen. La primera formada por el nombre de la operación en Java (definida por el atributo *signHere*), cuyos parámetros son términos. La respuesta de la operación es un valor de asertividad, es decir, *true* o *false*. La segunda instancia, lo constituye el mismo nombre de la operación en Java, pero seguido por la letra "t", y cuyo valor de retorno es un término.

La idea subyacente de la definición de dos tipos de instancias para la misma operación, radica en el hecho de la estructura misma de las clases Java. Como se recordará, en la sección 3.3.1 se discutió la pertinencia de las estructuras de la forma while-if. Las

instrucciones tipo “if” se utilizan para verificar la existencia de unificación entre términos, por lo cual, se requiere la devolución de valores booleanos. Pero, en el caso de términos con subtérminos en los cuales existan operaciones, la devolución de la operación debe ser un término, y no un valor booleano. No todas las operaciones requieren esta doble definición.

Las operaciones definidas dentro de la clase *Operators*, hasta el momento abarcan una cantidad de 16. En la tabla siguiente se muestra una descripción de las operaciones implementadas

Operación	Descripción
IS	Asignación numérica a una variables
SUMA	Realiza la suma de dos términos
RESTA	Ejecuta la operación de resta de dos términos
MAYOR	Devuelve un valor de verdad dependiendo si un término A es mayor a otro término B
MENOR	Devuelve un valor de verdad dependiendo si un término A es menor a otro término B
UNIV	Permite unificar listas a funtores y viceversa
UNIF	Devuelve un valor de verdad si puede unificar los términos
NUNIF	Devuelve un valor de verdad si no puede unificar los términos
OR	Convierte una operación or en Prolog a su equivalente en nuestra estructura de programa Java
IDENT	Devuelve un valor de verdad si los términos son idénticos
NIDENT	Devuelve un valor de verdad si los términos no son idénticos
NOT	Funciona como una operación not tradicional
ATOM	Devuelve un valor de verdad si el término es un átomo
CLAUSE	Devuelve el valor de verdad si es una cláusula
IFTHEN	Convierte una instrucción A;E -> G como A if E else G

TABLA 3.C. OPERADORES DEFINIDOS EN LA PROPUESTA

Una descripción más detallada de las operaciones y formato de las mismas, puede verse en los documentos de la API, en el anexo A. En la figura 3.17, se muestra en un diagrama UML la clase *Operators*.

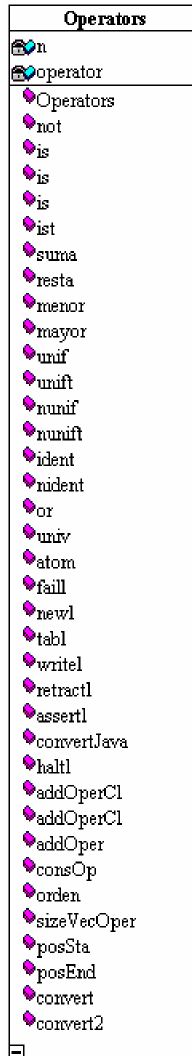

FIGURA 3.17. DIAGRAMA UML DE LA CLASE OPERADOR

Una de la razones de definir una clase individual de operadores (*Operators.class*) radica en la posibilidad de extender la riqueza de los métodos a través de nuevas operaciones. Esto podría ser muy útil en el caso, por ejemplo de CLP (Constraint Logic Programming). En [CLP-1] se plantea un mecanismo de prueba para CLP a través de mecanismos de inducción, los axiomas de pre-condición y post-condición éstos podrían definirse en la clase *Operators*. En [CLP-2] se discute las condiciones que debe ser tomadas en cuenta para la implementación de CLP basados en WAM o por medio de código generado en C. Adicionalmente, concluye que una de las formas más eficientes de implementar CLP es a través de la generación del código en C o C++ (claro esta bajo la limitante de un tiempo compilación alto). Estas pruebas se realizaron con diferentes implementaciones.

3.3.5. GENERACION DE CLASE DE CONSULTA

Al igual como en el caso un compilador de Prolog tradicional, se debe incorporar una interfaz para permitir la formulación de preguntas al programa a Prolog. Como se mencionó anteriormente, una de las consideraciones para la generación de clases Java por medio del programa realizado, consistió en colocar dentro de los métodos la información necesaria para que se puedan llevar a cabo la búsqueda de soluciones, sin la necesidad de un compilador o intérprete de Prolog como en el caso de [INT-1][JIN-1][COD-1].

Ahora bien, para poder llevar a cabo la consulta sobre un determinado programa convertido a Java, se requiere la definición de una clase en la cual se instancia el predicado asociado con los valores interrogados.

Se desarrolló un programa que genere la clase asociada a la pregunta de forma automática, este programa se denominó *GenerarQueries* ó *GenQ*, en modo gráfico y modo texto, respectivamente. La entrada a este programa consiste en un predicado que representa la consulta. Actualmente, solo se permite un predicado por consulta. En la figura 3.19 se muestra el diagrama UML del generador de consultas.

El proceso de generación de las consultas implica la utilización de los mecanismos de *parsing* discutidos en la sección 3.3.2. A partir de allí, se creó una nueva clase denominada *ScriptQ,* que genera la clase relacionada con la consulta. Siempre se generará la clase con el nombre de *Query.java* para almacenar la estructura de la consulta en Java. La estructura de la clase generada se muestra a continuación:

```
/* Supoganse que se realiza una consulta de la forma termN(term1, term2, …, termi) */
class Query{
   public static void main(String args[]) {
       TermN_aridadTermN  termN = new TermN_aridadTermN();
       if(termN.searchT(new Term(term1.name, term1.arity), new Term(term2.name,
       term2.arity), …, new Term(termi.name, termi.arity), CBackT )){
              System.out.println("//-------------- YES  ------------");
              System.out.println(termN.printAnswer(termN.termA1, new Term(term1.name,
              term1.arity)));
               …
              System.out.println(termNm.printAnswer(termNm.termAn,new
              Term(termn.name, termn.arity)));
       } else {
              System.out.println("//-------------- NO  ------------");
       }
   }
}
```

FIGURA 3.18. CODIGO DE UNA CLASE DE CONSULTA

De la figura anterior se desprende que la clase *Query.java,* consiste de una instancia de la clase asociada al predicado que se consulta. Adicionalmente, se procede al llamado del método de búsqueda soluciones denominado *searchT.*

Para aclarar un poco mejor la estructura de la clase se presenta el diagrama UML correspondiente al programa de generación de consultas.

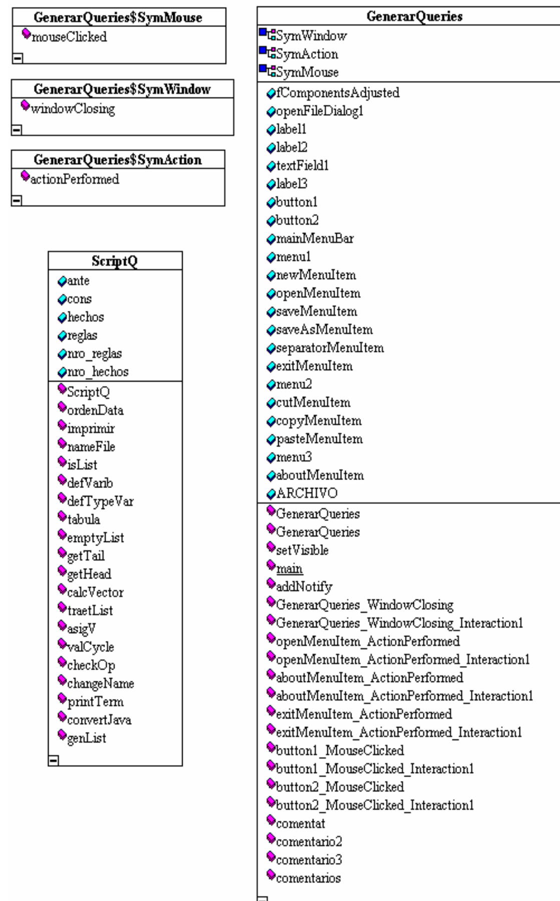

FIGURA 3.19. DIAGRAMA DEL PROGRAMA GENERADOR DE CONSULTAS

En el siguiente capítulo, se describirá el proceso de consulta a un programa converso mediante un ejemplo práctico.

Para realizar una consulta al programa, se debe generar una clase *Query.java*. Se generó una clase denominada *GenerarQueries*. La entrada al programa debe ser un predicado con los terminos a consultar, por ejemplo sea el predicado:

nuevo(A, B, C).

En la figura 3.20 se muestra la forma de realizar la llamada al programa generador de consultas. En la figura 3.21 se puede observar el código generado por la consulta.

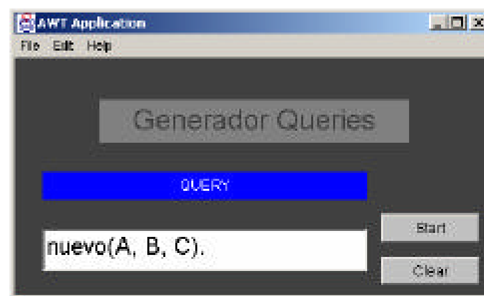

FIGURA 3.20. PROGRAMA GENERADOR DE CONSULTAS.

```
import progtojav.*;
import java.util.*;

/**
* <p>Definicion de la clase asociada a las queries</p>
* @author Jhon Edgar Amaya
* @version 1.2
*/

class Query{

        public static void main(String args[]) {

                Nuevo_3 nuevo = new Nuevo_3();
                if( nuevo.searchT(new Term("A", 0),new Term("B", 0),new Term("C", 0), 1)){
                   System.out.println("//-------------- YES  ------------");
                   System.out.println(nuevo.printAnswer(nuevo.termA1,new Term("A", 0)));
                   System.out.println(nuevo.printAnswer(nuevo.termA2,new Term("B", 0)));
                   System.out.println(nuevo.printAnswer(nuevo.termA3,new Term("C", 0)));
                } else {
                   System.out.println("//-------------- NO  ------------");
                }
        }

} //fin de clase
```

FIGURA 3.21. CLASE DE CONSULTA ASOCIADA AL PREDICADO NUEVO/3.

En la próxima sección se discutirán las características principales de la arquitectura planteada en [DAV-1] para la programación de agentes. Adicionalmente, se expondrá como se adaptan los mecanismos de traducción discutidos en las tres primeras secciones del presente capítulo con la arquitectura del agente. Por último, se abordarán las clases adicionales creadas para facilitar al programador la generación de definiciones y restricciones propias del agente.

3.4. LA PLATAFORMA DEL AGENTE

Un agente sensa y actúa, pero también realiza alguna forma de razonamiento para relatar percepciones a acciones y obtener su propia agenda. Un agente, debe ser visto como un sistema intencional: un sistema con intenciones y aptitudes tales como metas y creencias [WOO-4]

Dávila basándose en una propuesta realizada por Robert Kowalski, la cual, básicamente prescribe el uso de programación lógica para especificar agentes que son tanto reactivos como racionales; desarrolló, una propuesta que representa una especificación completa que incluye, la especificación de un procedimiento de prueba que sea utilizado como un mecanismo de razonamiento del agente. Esta especificación es totalmente trasladable, en un programa lógico, y es por lo tanto una especificación ejecutable [DAV-1]

La tabla 3.d muestra el formato principal de la especificación, que ha sido llamado GLORIA (el acrónimo de *General-purpose, Logic-based, Open, Reactive and Intelligent Agent*), la razón de este nombre es que se espera tratar diferentes especificaciones en el futuro.

El principal componente en la descripción lógica de Kowalski de un agente es un predicado meta-lógico, conocido como el predicado CYCLE, el cual ha sido transformado en la definición de la arquitectura del agente GLORIA [DAV-1].

El ciclo de Gloria se especifica en lógica de primer orden. Las especificaciones se pueden resumir como sigue:

- El ciclo de predicado, [GLOCYC], describe un proceso por el cual el estado interno cambia, mientras el agente asimila entradas desde el ambiente y envía las salidas al ambiente, después de consumir los "fragmentos" de tiempo definidos para el razonamiento. El proceso de razonamiento es modelado por el predicado DEMO y el mecanismo de sensar-actuar es modelado por el predicado ACT.

- El predicado ACT ([GLOACT] y [GLOEXE]) explica como el agente selecciona todas sus acciones planificadas que seria ejecutadas en un momento particular, y enviarlas en paralelo, al ambiente. El ambiente, de acuerdo al predicado TRY, admitirá estas acciones, y tratara de ejecutarlas simultáneamente, respondiendo de vuelta al agente con el resultado. Esta realimentación tiene la forma de hechos

observacionales que pueden ser agregados a la base de conocimiento del agente para su posterior procesamiento.

Observar los argumentos para el predicado DEMO. Este predicado reduce las metas a nuevas metas para ser considerados en su base de conocimiento. DEMO es, precisamente, la incorporación de la definición de “creencias”, la relación entre el agente y sus creencias. La palabra proviene de la frase “Un agente cree que puede DEMOstrar”.

Lo anterior puede explicar perfectamente los primeros tres argumentos de DEMO. El cuarto argumento de DEMO, es importante ya que indica la cantidad de recursos o de tiempo disponibles para el razonamiento. En cada ciclo, el agente razona para una cantidad limitada de tiempo, con la intención de prevenir ciclos infinitos de búsqueda. Esto permite implementar una importante característica de los agentes reales: no puedes pensar por siempre sin actuar.

La concepción antropomórfica de un agente es útil porque permite la descripción de una entidad que es autónoma en la consecución de objetivos para los cuales ha sido programado. Bajo este argumento, justo como los objetos son una abstracción poderosa que soporta el desarrollo de sistemas computacionales, los agentes son muy adecuados para el desarrollo de sistemas computacionales complejos. Uno de los elementos claves en esta concepción es que los desarrolladores de sistemas basados en agentes puedan concentrarse en descripciones de alto nivel de la estrategia conocida como "know-how" y los objetivos para los agentes, dejando para el final decisiones acerca cuando deben hacerlo.

La arquitectura del agente de [DAV-1] utiliza un procedimiento básico denominado Demo, como se menciono anteriormente; para llevar a cabo la realización de los mecanismos de procesamiento de datos provenientes del exterior, generando los planes de las acciones a seguir por parte del agente.

GLORIA	
cycle (KB, Goals, T) **demo (KB, Goals, Goals´, R)** **^R < n** **^act (KB, Goals´, Goals´´, T+R)** **^cycle (KB, Goals´´, T+R+1)**	**[GLOCYC]**
act (KB, Goals, Goals´, T_a) **Goals PreferredPlan V AltGoals** **^executables (PreferredPlan, T_a, TheseActions)** **^try (TheseActions, T_a, Feedback)** **^assimilate (Feedback, Goals, Goals´)**	**[GLOACT]**
executables (Intentions, T_a, NextActs) **forall A,T (do(A,T) is_in Intentions** **^ (consistent((T=T_a) ^ Intentions)** **do (A,T_a) is_in NextActs))**	**[GLOEXE]**
assimilate (Inputs, InGoals, OutGoals) **forall A, T´,T´´ (action(A, T, succeed) is_in Inputs** **^do(A,T´) is_in InGoals do(A,T) is_in NGoal)** **^ forall A, T´,T´´ (action(A,T,fails) is_in inputs** **^do(A,T') is_in InGoals (false do(A,T')) is_in n Goals** **^ forall P, T´ (obs (P,T) is_in Inputs** **^Item action(A,T,R)** **^not (obs (P,T) is_in InKB) obs(P,T) is_in NGoal)** **^forall Atom (Atom is_in NGoals Atom is_in Inputs)** **^OutKB = Observed ^ InKB** **^OutGoals NGoals ^InGoals**	**[GLOASSI]**
A is_in B B A ^ Rest	**[GLOISN]**
try(Output, T, Feedback) **Tested on and by the environment…**	**[TRY]**

TABLA 3.D. DESCRIPCION DE GLORIA. TOMADO DE [DAV-1]

La máquina de inferencia se basa en doce cláusulas principales que utilizan el predicado demop/4. En la tabla 3.e, se muestran las 12 cláusulas [DAV-3].

1	**demop(G, G, [], 0).**
2	**demop(goals(true, R), R, [], _).**
3	**demop(goals(splan(if((A, B), C), RP), RG), NGoals, Influences, R) :- atom(A), in(A, RP), NR is R - 1, demop(goals(splan(if(B, C), RP), RG), NGoals, Influences, NR).**
4	**demop(goals(splan(if((A, B), C), RP), RG), NGoals, Influences, R) :- atom(A), definition(A, Def), NR is R - 1, demop(goals(splan(if((Def, B), C), RP), RG), NGoals, Influences, NR).**
5	**demop(goals(splan(if(true, C), RP), RG), NGoals, Influences, R) :- NR is R - 1, demop(goals(splan(C, RP), RG), NGoals, Influences, NR).**
6	**demop(goals(splan(if((or(A, B), C), D), RP), RG),NGoals,Influences, R):- NR is R - 1, demop(goals(splan(if((A, C), D), splan(if((B, C), D), RP)), RG), NGoals, Influences, NR).**
7	**demop(goals(splan(or(A, B), RP), RG), NGoals, Influences, R) :- agregar_plan(A, RP, NA), agregar_plan(B, RP, NB), NR is R - 1, demop(goals(NA, goals(NB, RG)), NGoals, Influences, NR).**
8	**demop(goals(splan(A, RP), RG), NGoals, Influences, R) :- atom(A), in(A, RP), NR is R - 1, demop(goals(RP, RG), NGoals, Influences, NR).**
9	**demop(goals(splan(A, RP), RG), Ngoals, Influences, R) :- atom(A), definition(A, Def), NR is R - 1, demop(goals(splan(Def, RP), RG), NGoals, Influences, NR).**
10	**demop(goals(splan(A, RP), RG), NGoals, [A| Influences], R) :- atom(A), executable(A), NR is R - 1, demop(goals(RP, RG), NGoals, Influences, NR).**
11	**demop(goals(splan(A, RP), RG), Ngoals, Influences, R) :- atom(A), observable(A), NR is R - 1, demop(goals(RP, RG), NGoals, Influences, NR).**
12	**demop(G, G, [], _).**

TABLA 3.E. CLAUSULAS DE PRINCIPALES DE DEMO. TOMADO DE [DAV-1]

A continuación se procederá a describir cada una de ellas de las instrucciones mostradas en la tabla 3.E [DAV-3].

La primera cláusula determina la finalización del proceso de razonamiento cuando los recursos se han agotado. Como se discutió anteriormente, la idea de este parámetro es evitar que el agente permanezca en un estado de razonamiento *ad infinitum*. La segunda cláusula permite detener el proceso de razonamiento una vez sea completado algún plan por parte del agente.

La tercera cláusula permite la propagación de las diferentes definiciones. La cláusula cuatro permite desglosar los elementos constitutivos de las definiciones. Las cláusulas cinco, seis, siete, ocho y nueve, permiten la manipulación de las diferentes equivalencias y la simplificación de términos.

Las cláusulas diez y once, permiten la "consumir" las observaciones del medio, es decir que el agente puede procesar cada uno de los términos asociados a las observaciones provenientes del ambiente. Adicionalmente, se pueden producir influencias, que representan las posibles acciones que podría llevar a cabo el agente.

La última cláusula indica que no se puede hacer nada acerca del proceso de razonamiento, es decir es una cláusula de control.

3.4.1. EL AGENTE Y EL MECANISMO DE TRADUCCION

En el capítulo 2, se discutió la definición de agentes y sus posibles arquitecturas, formas de programación, así como, algunos puntos de vista sobre la integración de lenguajes declarativos y orientados a objetos. ¿Pero será adecuados estos paradigmas como plataforma de diseño de agentes inteligentes?. Una importante afirmación acerca de la integración de lenguajes declarativos y lenguajes orientados a objeto para la programación de agentes, la brinda Morozov [MOR-1] que indica que:

> *"La idea de un objeto... [como base] de las aplicaciones orientadas a objetos no tiene una implementación univalente en lógica. Todos los intentos por transferir esta idea en lenguajes lógicos, presentan dificultades al menos en tres aspectos...*
>
> *[En primer lugar los] aspectos estructurales de la OOA[29], los objetos en la programación imperativa tienen significado natural en la estructura del programa. En la programación lógica este aspecto se refiere al programa lógico estructurado textualmente y a las facilidades para el control del espacio de búsqueda.*
>
> *[Como segunda instancia] el aspecto dinámico de la OOA, refleja las posibilidades relacionadas a la modificación de los objetos en la OOA, imperativa. Este aspecto causa las más grandes dificultades en la teoría de programación lógica; existen problemas muy fuertes de integración de objetos con la programación lógica, como por ejemplo persistencia vs. "backtrackable-state"*
>
> *[Por ultimo] el aspecto de información de la OOA, esta asociado con el problema de descripción de estructuras de datos complicadas".*

[29] OOA. Acrónimo de Agentes orientados a objeto

Adicionalmente [BAN-1] menciona que existe un beneficio de las acciones declarativas sobre el paradigma imperativo, ya que podría permitir al desarrollador del sistema, - en nuestro caso el programador del agente -, tratar un sistema complejo como una colección de componentes que cooperan. Propone además que los agentes diseñados con ambas filosofías de diseño no deberían ser vistos como constituidos por una simple interfaz que permita separar los aspectos lógicos de las facilidades imperativas. Desde nuestro punto de vista, y tomando como base las afirmaciones de Morozov y Calejo en cuanto a la integración de ambos sistemas, seria perfectamente válido definir un punto de diseño que permita separar los aspectos declarativos e imperativos para la programación de agentes, lo cual es diametralmente opuesto a lo expresado en [BAN-1].

Ahora bien, para realizar la llamada al agente se puede hacer de dos formas básicas:

La primera consiste, en utilizar el programa Prolog que se anexa al paquete de traducción, ubicado en la carpeta *InfeEng* denominado *InfeEng.pro.* Luego, se procede a realizar la conversión del lenguaje Prolog a Java, mediante las técnicas descritas anteriormente; con lo cual, se generan los archivos fuentes *.java* correspondiente a los predicados asociados al agente y representados por la función principal denominada *demo*. Posteriormente se debe realizar la compilación los archivos *.java* para generar los archivos correspondientes *.class*

Posteriormente, debe realizarse el mismo procedimiento para la generación de la base de conocimiento, que la podemos representar en cuatro predicados propios de demo, a saber: *ic/2, obs/1, definition/2* y *executables/1.* En la carpeta *InfeEng* del compendio de clases de traducción, se encuentra un archivo base denominado *KBInfeEng.pro*, con algunos predicados de *ic/2, obs/1, definition/2* y *executables/1.*

La segunda forma consiste en definir un único archivo donde se escriban tanto los predicados asociados a la máquina de inferencia como las restricciones y definiciones.

En la siguiente figura podemos realizar una descripción gráfica de cómo se relacionan estos predicados y la función *demo* con la definición de agente según Russell.

Como se mencionó en el capítulo 1, el agente esta basado en el ciclo de Kowalski. El agente representado por la función demo, lleva a cabo la generación de posibles planes que podrían ejecutarse dependiendo de la entrada observada y luego del proceso de razonamiento por medio del ciclo y la base de conocimiento. El predicado obs, tiene relación con los sensores del agente. El predicado executable son todos aquellos predicados que pueden llevar a cabo el agente. Los predicados definition e ic, permiten

definir las reglas de condición-acción del agente así como las restricciones correspondientes para evitar acciones inconsistentes.

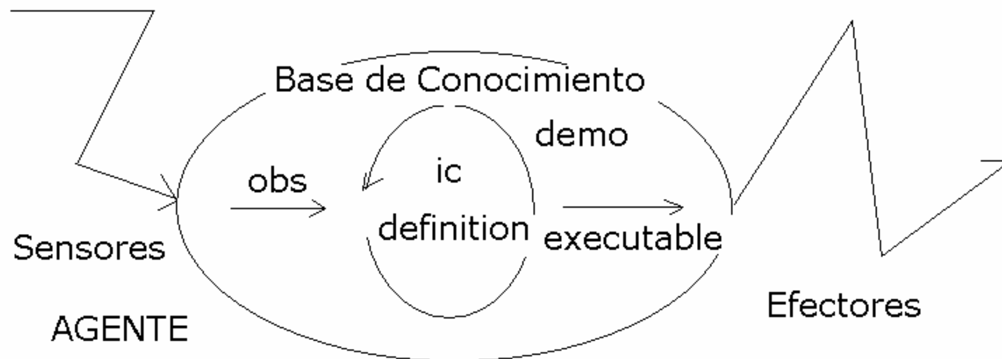

FIGURA 3.22. DESCRIPCION PICTORICA DEL AGENTE

Un ejemplo de la invocación del agente se muestra a continuación

%Invoking demo
demo(Gin, Gout, Influences). *(3.15)*

En [DAV-1], describe una plataforma que utiliza la programación lógica de forma generalizada para la programación de agentes que integren características tales como, reactividad, apertura, así como las nociones de creencias, objetivos y actividades mentales.

Esta plataforma esta constituida por cuatro lenguajes de programación, y representa una amalgama entre objetos y programas meta-lógicos para el modelado de conceptos, tales como, creencias, objetivos y actividades mentales, con el fin de brindar un modelo más realista del agente.

Estos predicados, junto con otros que realizan operaciones de manipulación de datos, que se omiten de la discusión; se incluyen en un archivo con el fin de realizar la conversión de la máquina de inferencia a un formato en Java, utilizando el programa *GenClass.*

% java GenClass agente

Lo cual generara las clases que se muestra en la tabla.

Clases
Demop_4.java
Agregar_plan_2.java
In_2
Ic_2
Definition_2
Executable_1
Obs_1
Demo_3.java

TABLA 3.F. CLASES GENERADAS POR PROGRAMA TRADUCTOR

3.4.2. GENERADOR DE CLASES ESPECIALES PARA EL AGENTE

Como se mencionó en la sección anterior la información de la máquina de inferencia esta definida por el procedimiento de prueba denominado DEMO. La base de conocimiento sobre la cual realizara los procesos de inferencia esta determinada básicamente por la definición de dos predicados: *ic/2, definition/2*, además de dos predicados que expresan la posibilidad de sensar y actuar sobre el ambiente como lo son: *executables/1* y *obs/1*.

En la estructura de demo se puede observar que el predicado *definition/2*, se representa en lógica de predicados, esto significa que todos los términos allí incluidos no pueden contener variables, solo valores constantes que a su vez definen el dominio de Herbrand y como consecuencia el espacio de búsqueda de soluciones para el agente.

La forma de representar una definición para ser utilizada por el agente esta conformado por un predicado de la forma:

definition(condición, acciones).

Donde condición representa el Objeto que se esta procediendo a definir. En este caso Objeto es un fragmento de conocimiento para el agente, no necesariamente un objeto tangible.

Acciones, representa las operaciones que conlleva la definición. Estas acciones son del tipo if-then, con la posibilidad de incluir operaciones or. Adicionalmente como convención se incluyen las acciones en un predicado denominado splan/2. Por ejemplo, supóngase un agente que juega al béisbol, entonces podemos definir:

definition(dar_boleto_intencional,or(splan(batea_cuarto_bate, splan(bases_llenas, true), splan(batea_Barry_Bonds, true))). *(3.16)*

La definición anterior indica que el manager tiene por regla dar boleto intencional en dos tipo de ocasiones: solo si tiene las bases llenas y el cuarto bateador en *lineup* está al bate o en caso que el bateador sea Barry Bonds. Esta regla puede desglosarse en lógica clausal, como:

Dar_boleto_intecional :- (batea_cuarto_bate, bases_llenas); batea_Barry_Bonds.

Debido a este requerimiento, las reglas de forma clausal que el desarrollador de sistemas multiagentes define para la programación del agente requiere una conversión al formato a la lógica de predicados mostrados en (3.16).

Para lograr que este proceso sea lo más transparente posible para el programador se desarrollo un programa que realice forma automáticamente la conversión de la lógica de forma clausal de definiciones al formato requerido por el agente.

El programa aprovecha las clases definidas para realizar el proceso de parsing de las cláusulas, discutidas anteriormente. En primer lugar, se realiza la instancia de dichas clases y la salida se convierte en la entrada de una nueva clase que permite la generación de la conversión. Esta clase se conoce como *ScriptDef.*

ScriptDef se incluye en el programa denominado *GenerarDef.class* en su forma grafica o *GenDef* en modo texto. Un ejemplo, de cómo se ejecutaría el programa en (3.16). En primer lugar, creamos un archivo donde residen las definiciones, suponiendo en este caso Béisbol_Manager_Def.

Se puede luego utilizar alguno de los programas mencionados anteriormente. En el caso del programa tipo texto, se puede llevar a cabo de la siguiente forma:

%java GenDef nombre_archivo

En cualquiera de los dos formatos, el programa generara un nuevo archivo de nombre definitions. El resultado para el ejemplo anterior es:

```
%* Conversion de la 'definition'
%* @author Jhon Edgar Amaya
%* @version 1.0
definition(dar_boleto_intencional, or(splan(batea_cuarto_bate, splan(bases_llenas, true),
splan(batea_Barry_Bonds, true))).
```

Una vez se ha realizado la conversión se puede utilizar el programa de generación de clases para obtener la clase definition, necesaria para el proceso de funcionamiento de la máquina de inferencia del agente.

Adicionalmente al predicado de *definition/2*, se encuentra el predicado de ic/2. Este predicado se conoce como constantes de restricción. La idea de este predicado es evitar el surgimiento de estados insistentes, dentro del proceso de razonamiento del agente. Por ejemplo,

En el caso del ejemplo (3.16), podríamos incluir una constante de restricción siguiente:

ic(splan(if((batea_equipo_contrario), dar_boleto_intencional), true)). (3.17)

Esta cláusula indica que solo si el equipo contrincante esta al bate se puede dar un boleto intencional. Puedes observarse que esta cláusula corresponde a la expresión:

If batea_equipo_contrario then dar_boleto_intencional (3.18)

(3.17) constituye la forma en que se definen las constantes de restricción de la máquina de inferencia en la plataforma de agentes desarrollada por Dávila [DAV-1]. Nosotros proponemos la utilización de una escritura más cercana a la sintaxis de Prolog, es decir, si tenemos reglas de la forma *if condición then acción*, la escribiéramos de forma *acción :- condición* [HOG-1]. Entonces (3.18) se convertiria en:

Dar_boleto_intencional:- batea_equipo_contrario.

El programador de la base de conocimiento del agente escrito de esta forma debe traducirse de tal forma que el procedimiento de prueba pueda manipularlo, se requiere la creación de un programa que lo lleva a cabo.

Se definió un programa en dos versiones modo grafico y texto que llevara a cabo el mecanismo de ajuste a la sintaxis requerida por el mecanismo de inferencia. El programa se denomina *GenerarIC* ó *GenIC* según sea el caso. El programa hace uso de las clases definidas para realizar el proceso de parsing de las cláusulas, discutidas con anterioridad. En primer lugar, se realiza la instancia de dichas clases y la salida se convierte en la entrada de una nueva clase que permite la generación de la conversión. Esta clase se conoce como *ScriptIC*.

ScriptIC se incluye en el programa denominado *GenerarIC.class* en su forma grafica o *GenIC* en modo texto. Un ejemplo, de cómo se ejecutaría el programa (3.16). En primer

lugar, creamos un archivo donde residen las restricciones, en este caso Béisbol_Manager_IC.

Se puede luego utilizar alguno de los programas mencionados anteriormente. En el caso del programa tipo texto, se puede llevar a cabo de la siguiente forma:

```
%java GenIC nombre_archivo
```

En cualquiera de los dos formatos, el programa generara un nuevo archivo de nombre integCons. El resultado para el ejemplo anterior es:

```
%* Conversion de las 'restricciones'
%* @author Jhon Edgar Amaya
%* @version 1.0
ic(splan(if( ( batea_equipo_contrario), dar_boleto_intencional), true) ).
```

Luego se puede proceder a utilizar el programa para la generación de clases, y obtener de esta forma la clase para el agente.

CAPÍTULO 4. PRUEBAS Y RESULTADOS

4. PRUEBAS Y RESULTADOS

Una vez determinado el procedimiento de traducción de Prolog a Java, y evaluado el funcionamiento de la arquitectura de programación de agentes planteada en [DAV-1] y analizado el potencial de su traducción en Java, se procedió a la generación del código correspondiente a la propuesta, tal como se describió en el capítulo 3. Luego de último paso, era necesario definir escenarios de prueba para llevar la evaluación del diseño propuesto. Para tal fin, se llevaron a cabo diferentes experimentos para comprobar la efectividad de los mecanismos de traducción planteados en el proyecto. Podemos reunir los experimentos en tres bloques específicos. El primero que consiste en el desarrollo de programas "ligeros", para corroborar el correcto funcionamiento de los mecanismos básicos que deben presentar los lenguajes declarativos pero ahora bajo la óptica de clases de java. El segundo consiste en la traducción de programa de la máquina de inferencia, conocida como "demo" y la comprobación de su correcto funcionamiento mediante la inclusión de reglas simples. Por ultimo, se desarrollo una prueba del mecanismo de inferencia del agente para un proyecto específico como lo es Bioinformantes, donde se incluyen reglas más densas en cuanto a cantidad, así como en complejidad.

4.1. PRUEBAS BÁSICAS

A continuación, se describiran las pruebas básicas realizadas para verificar el funcionamiento de nuestra propuesta. Se llevaron a cabo, tres conjuntos de banco de pruebas. El primero, verifica el funcionamiento de la resolucion y la asertividad de las consultas. El segundo conjunto, permite comprobar el soporte de la multimodalidad y la negación por falla. Para el último banco de pruebas básicas, diseñamos un programa para corroborar la correctitud de las operaciones aritméticas.

4.1.1. RESOLUCION Y CONSULTAS.

En primer lugar, para comprobar el funcionamiento de los mecanismos de traducción diseñados se procedió a realizar un programa tradicional en Prolog para la resolución de genealogías.

Definimos los predicados correspondientes al programa en Prolog y procedemos a almacenarlos en un archivo, en este caso *Gen.pro*, cuyo contenido se muestra a continuación.

```
% Programa de Gen.pro

hijo(jhon, edgar).
hijo(leo, edgar).
hijo(carlos, edgar).
hijo(jhon, carmen).
hijo(leo, carmen).
hijo(carlos, carmen).

fem(carmen).

masc(edgar).
masc(jhon).
masc(leo).
masc(carlos).

hermano(A, D) :- hijo(A, Z), hijo(D, Z), not(A = D).

madre(F,G) :- hijo(G,F), fem(F).

padre(F,G) :- hijo(G,F), masc(F).                                  (4.1)
```

Para realizar la conversión en las clases correspondientes en Java, se procedió a ejecutar el programa *GenClass* (o en su defecto, se puede utilizar el programa *GenerarClases*, que es una interfaz gráfica que permite introducción de los parámetros de conversión mediante el uso de ventanas tipo AWT); se introduce como parámetro el archivo *Gen.pro*. Es decir, el llamado del programa se ejecutaría de la siguiente forma:

```
%java GenClass Gen.pro
```

Al ejecutar el programa, se generaron los siguientes archivos con la estructura discutida en el capítulo 3.

Hijo_2.java
Fem_1.java
Masc_1.java
Hermano_2.java
Madre_2.java
Padre_2.java

Puede observarse que para cada predicado se genera una clase java, con el nombre del predicado seguido de un “underscore” y la aridad asociada al predicado, tal como se discutió en el capítulo anterior.

Para realizar una consulta al programa se debe generar una clase que contenga la pregunta que queremos responder. Por ejemplo, una consulta sencilla seria saber de quien es hijo “leo”, esta consulta corresponde a la pregunta *?hijo(leo, G)*. El predicado se introduce en el programa *GenerarQueries*, tal como se muestra en la figura 4.1.

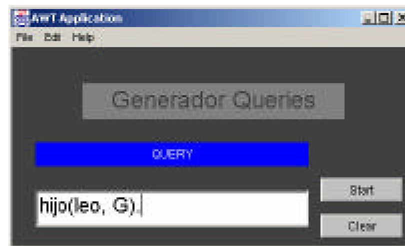

FIGURA 4.1. CONSULTA A UN PROGRAMA DE PRUEBA

Este programa genera una clase denominada *Query.java*, que sirve para que el usuario pueda realizar la consulta, tal y como se presenta a continuación:

```
import progtojav.*;
import java.util.*;

/**
* <p>Definicion de la clase asociada a las queries</p>
* @author Jhon Edgar Amaya
* @version 1.2
*/

class Query{

	public static void main(String args[]) {

		Hijo_2 hijo = new Hijo_2();
		if( hijo.searchT(new Term("leo", 0),new Term("G", 0), 1)){
			System.out.println("//-------------- YES  ------------");
			System.out.println(hijo.printAnswer(hijo.termA1,new Term("leo", 0)));
			System.out.println(hijo.printAnswer(hijo.termA2,new Term("G", 0)));
		} else {
			System.out.println("//-------------- NO  ------------");
		}
	}

}
```

Procedemos a compilar la clase *Query* y ejecutamos el *.class* correspondiente, obteniéndose la respuesta a la consulta.

```
//-------------- YES  ------------
::: leo
::: G/edgar
```

Puede concluirse que la consulta realizada sobre el clase *Hijo_2* fue exitosa al obtenerse el valor deseado, tal y como se llevaría a cabo en un IDE[30] de Prolog, como por ejemplo Amzi!.

Cabe destacar, que utilizamos la palabra “exitosa”, para asociar el hecho de que dadas las diferentes cláusulas del programa, hubo una forma de encontrar una solución a la consulta planteada. Esto último se puede ratificar con la presencia de la cadena:

```
//-------------- YES  ------------
```

El resultado se puede interpretar como que *“leo”* tiene por padre a *“edgar”*. Es de observar, que el término *padre/2* se refiere al de progenitor, sin distingo de género.

4.1.2. MULTIMODALIDAD Y NEGACION POR FALLA.

Una vez comprobada la posibilidad de realizar consultas simples, donde solo se ha involucrado una clase y sus respectivos métodos. Se procedió a realizar consultas un poco más complejas, en las cuales se hacen llamados a otros métodos de otras clases diferentes a la de clase consultada, con la intención de probar los mecanismos de resolución de reglas. Por ejemplo, se puede consultar si existe un par de hermanos dentro del programa Prolog de genealogía descrito anteriormente. Esto se traduciría como un sentencia: *hermano(W, G),* donde G y W son dos variables arbitrarias.

Utilizando el mismo procedimiento descrito anteriormente se procedió a realizar una nueva consulta utilizando el generador de *queries*, obteniéndose:

```
//-------------- YES  ------------
::: W/jhon
::: G/leo
```

[30] Acrónimo en inglés de Ambiente de desarrollo interactivo.

La respuesta anterior nos indica que existe un par de hermanos: *"leo"* y *"jhon"*.

Para probar la multimodalidad de Prolog, es decir, dos consultas con diferente estructura pueden generar una misma respuesta. Ergo, se utilizaron dos consultas *hermano(jhon, X)* y *hermano(X, leo)*.

Se procedió a realizar la consulta *hermano(jhon,X)*, para ello de utilizó el programa *GenQ*, como ya se mencionó. El resultado arrojado fue:

```
//-------------- YES  ------------
::: jhon
::: X/leo                                                                    (4.2)
```

luego, se procedió a realizar la consulta *hermano(X, leo),* dando el siguiente resultado:

```
//-------------- YES  ------------
::: X/jhon
::: leo                                                                      (4.3)
```

Como se puede observar en (4.2) y (4.3), dos consultas diferentes pueden involucran idénticas respuestas, es decir, la construcción del espacio de búsqueda generado a partir de un conjunto de diferente de consultas puede conducir a hallar soluciones idénticas. Podemos concluir con este sencillo ejemplo, que nuestra propuesta maneja la multimodalidad que presenta un lenguaje como Prolog.

Retomando el programa (4.1), se puede observar la existencia el predicado *not*, que el caso de Prolog se maneja como cualquier otro predicado para permitir la negación por falla. En nuestra propuesta y siguiendo como base [SBP-1], se definió el predicado *not* como *built-in,* es decir, como un elemento esencial para funcionamiento del programa. Todos los *built-in* se agrupan en la clase *Operators*, como se discutió en el capítulo 3. De aquí en adelante llamaremos "operaciones" a todos aquellos predicados que se encuentran en la clase *Operators*, así como a las diferentes operaciones clásicas.

De nuevo en el programa (4.1) se utiliza el predicado *not* anidado con una operación de unificación "=". Este predicado en el programa evita las respuestas incongruentes, tales como, que *"jhon"* es hermano de si mismo. Por ejemplo, si se ejecuta la consulta *hermano(jhon, jhon),* la salida del programa es:

```
//-------------- NO  ------------
```

Esto último significa, que en todo el conjunto de reglas que componen el programa fue imposible encontrar o derivar el término *hermano(jhon, jhon).*

Ahora bien, si se modifica la regla *"hermano(A, D) :- hijo(A, Z), hijo(D, Z), not(A = D)."* del programa (4.1), omitiendo la totalidad del término que abarcan la operación *not*, es decir, convertir la regla en *"hermano(A, D) :- hijo(A, Z), hijo(D, Z).".* Podriamos realizar esta modificación de tres formas, primero modificando el código en *Gen.pro*, generando las clases nuevamente y compilando de éstas últimas. Nosotros utilizamos otro procedimiento, colocamos la regla en un nuevo archivo de nombre *regla.pro*, y procedimos a generar la nueva clase *Hermano_2,* y procedimos a compilar ésta última. La tercera posibilidad consiste en eliminar el siguiente código de la clase *Hermano_2* y proceder a compilarla:

```
public boolean searchRule(Term X1, Term X2, int CBackT){
        ...
         if (hijo_2i.searchT(H2,H3,CBackT)== true){
                 H2.asigValue(hijo_2i.termA1);
                 H3.asigValue(hijo_2i.termA2);
                /*
                Se elimina la siguiente línea
                */
                 if (oper.not(oper.unif(H1,H2))== true){
                        retorno = true;
                        ...
                        return retorno;
                /*
                Se elimina la siguiente línea
                */
                 }
         }
        ...
 } // fin de metodo
```

Luego, procedemos a realizar la consulta *hermano(W,G)* obtenemos:

```
//-------------- YES  ------------
::: W/jhon
::: G/jhon
```

Lo cual concuerda perfectamente con el funcionamiento de los compiladores de Prolog.

4.1.3. OPERACIONES ARITMETICAS

Para comprobar los mecanismos de manipulación aritmética, se diseñó un programa que calcula la sumatoria de números enteros. Sea el número entero i, entonces la sumatoria seria,

i + (i-1)+ (i-2) + ….+ 1.

Existe una fórmula general para la resolución del problema, es decir, dado un entero i, el resultado de la suma de sucesión de números enteros será:

```
S = (i * (i+1) ) / 2.
```

El programa en Prolog esta compuesto por dos cláusulas:

```
s(1,1).
s(F,G):- Z is F-1, s(Z,H) , G is H+F.
```

Procedemos de forma similar generando las clases correspondientes. En este caso solo se generará una clase denominada *S_2.java*.
Para comprobar el funcionamiento de la sumatoria, se generá una consulta del tipo:

```
s(10,G).
```

Al compilar la pregunta y proceder a ejecutar *Query.class*, arroja la respuesta:

```
//-------------- YES  ------------
::: 10
::: G/55
```

Podemos realizar una consulta diferente, por *s(20,F).* Realizamos el procedimiento para la generacion de consultas, obteniendose la siguiente respuesta,

```
//-------------- YES  ------------
::: 20
::: F/210
```

El programa anterior permite demostrar el funcionamiento de las operaciones aritméticas de suma y resta. Se requieren más pruebas para corroborar el funcionamiento de todo el universo de operaciones aritméticas sobre la propuesta, pero no hay razones para dudar de los cálculos aritméticos similares. A continuación podemos observar algunos fragmentos importantes del código de la clase *S_2.java*.

```
…
     T1.asigValue(H1);
     T2.asigValue(H2);
     argUf = eleUnig.formVector(T1,T2);
     mGUnf = eleUnig.unify(argum, argUf, arity);

    if (! eleUnig.getCondUnify()){
            varbl = eleUnig.formVector(H1,H2,H3,H4);
```

```
		varbl = eleUnig.buildSusVV(varbl,mGUnf);
		H1 = (Term) varbl.elementAt(0);
		H2 = (Term) varbl.elementAt(1);
		H3 = (Term) varbl.elementAt(2);
		H4 = (Term) varbl.elementAt(3);
		int i1=0;
		while(i1==0 || i1 < getNumberTerms()){
			if (oper.is(H3,oper.resta(H1,new Term("1", 0)))== true){
				H3.asigValue(oper.resta(H1,new Term("1", 0)));

				if (searchT(H3,H4,(CBackT+1))== true){
					H3.asigValue(termA1);
					H4.asigValue(termA2);
					if (oper.is(H2,oper.suma(H4,H1))== true){
						H2.asigValue(oper.suma(H4,H1));
						retorno = true;

						varbl = eleUnig.formVectorSpec( conVarLi(
						CBackT, 1, contBack), H1, conVarLi( CBackT, 2,
						contBack), H2, conVarLi( CBackT, 3, contBack),
						H3,conVarLi(CBackT,4,contBack),H4);

						termA1 = eleUnig.buildSusTV(T1, varbl);
						termA2 = eleUnig.buildSusTV(T2, varbl);
						return retorno;
					}
				}
			}

			i1++;
		} //fin de ciclo i1
	}// fin del if
…
```

4.2. PRUEBAS DEL MOTOR DE INFERENCIA DEL AGENTE

Para verificar el correcto funcionamiento tanto de los mecanismos de traducción como de los elementos necesarios para el funcionamiento del motor de inferencia del agente, se procedió a la conversión del programa *demo* y se definió una base mínima de conocimiento formada por los predicados siguientes,

definition(portar_sombrilla, or(splan(buscarla, splan(asirla, true)), splan(prestarla, true))).
ic(splan(if((esta_lloviendo, true), portar_sombrilla), true)).
executable(buscarla).
executable(asirla).
executable(prestarla).
observable(esta_lloviendo).

El código anterior plantea la existencia de tres (3) acciones atómicas que podría llevar a cabo el agente y que son representadas por el predicado *executable/1*. El predicado *executable/1* le informa al metaprograma *demo.pl* que los términos "buscarla", "asirla" y "prestarla"; deben ser procesados como acciones en un plan. En forma similar, *observable/1* le dice a *demo.pl* que el término "esta_lloviendo" representa una observación. No significa que el agente haya observado hecho alguno. Significa que podria observar esas "propiedades" en su entorno.

Las definiciones es la forma de representar el conocimiento procedimental. Para hacer tarea alguna, es necesario definir esa tarea por medio del término en definition/2 e indicarle que ejecute un plan u otro definido en el término. Un plan es una colección de subplanes que se representa con el predicado *splan/2*. Cada subplan tiene una primera acción y el resto constituye el conjunto de acciones -que pueden no ser todavía un término *executable/1*-.

Se utilizan las restricciones para indicar las condiciones en que se activan ciertas tareas para el agente [DAV-1].

Se procedió a la generación de las clases asociadas al procedimiento de prueba denominado *"demo"*, incluyendo la base de conocimiento descrita anteriormente.

Para realizar la consulta al motor de inferencia se define una pregunta, en nuestro caso *demo(true, S, F),* y se introduce en el programa generador de consultas, obteniendo el archivo *Query.java* con el siguiente contenido.

```
import progtojav.*;
import java.util.*;
/**
* <p>Definicion de la clase asociada a las queries</p>
* @author Jhon Edgar Amaya
* @version 1.2
*/
class Query{
        public static void main(String args[]) {
                Demo_3 demo = new Demo_3();
                if( demo.searchT(new Term("true", 0),new Term("S", 0),new Term("F", 0), 1)){
                    System.out.println("//-------------- YES  ------------");
                    System.out.println(demo.printAnswer(demo.termA1,new Term("true", 0)));
                    System.out.println(demo.printAnswer(demo.termA2,new Term("S", 0)));
                    System.out.println(demo.printAnswer(demo.termA3,new Term("F", 0)));
                } else {
                    System.out.println("//-------------- NO  ------------");
                }
        }
}
```

El diagrama UML de las clases involucradas en el experimento se presentan a continuación. Se incluye la representación de las cláusulas asociadas a *"demo"* pero se omite su relación con las clases principales de funcionamiento en Java, como por ejemplo *Term*, por razones de espacio. Los predicados asociados a *"demo"* se discutieron en el capítulo anterior.

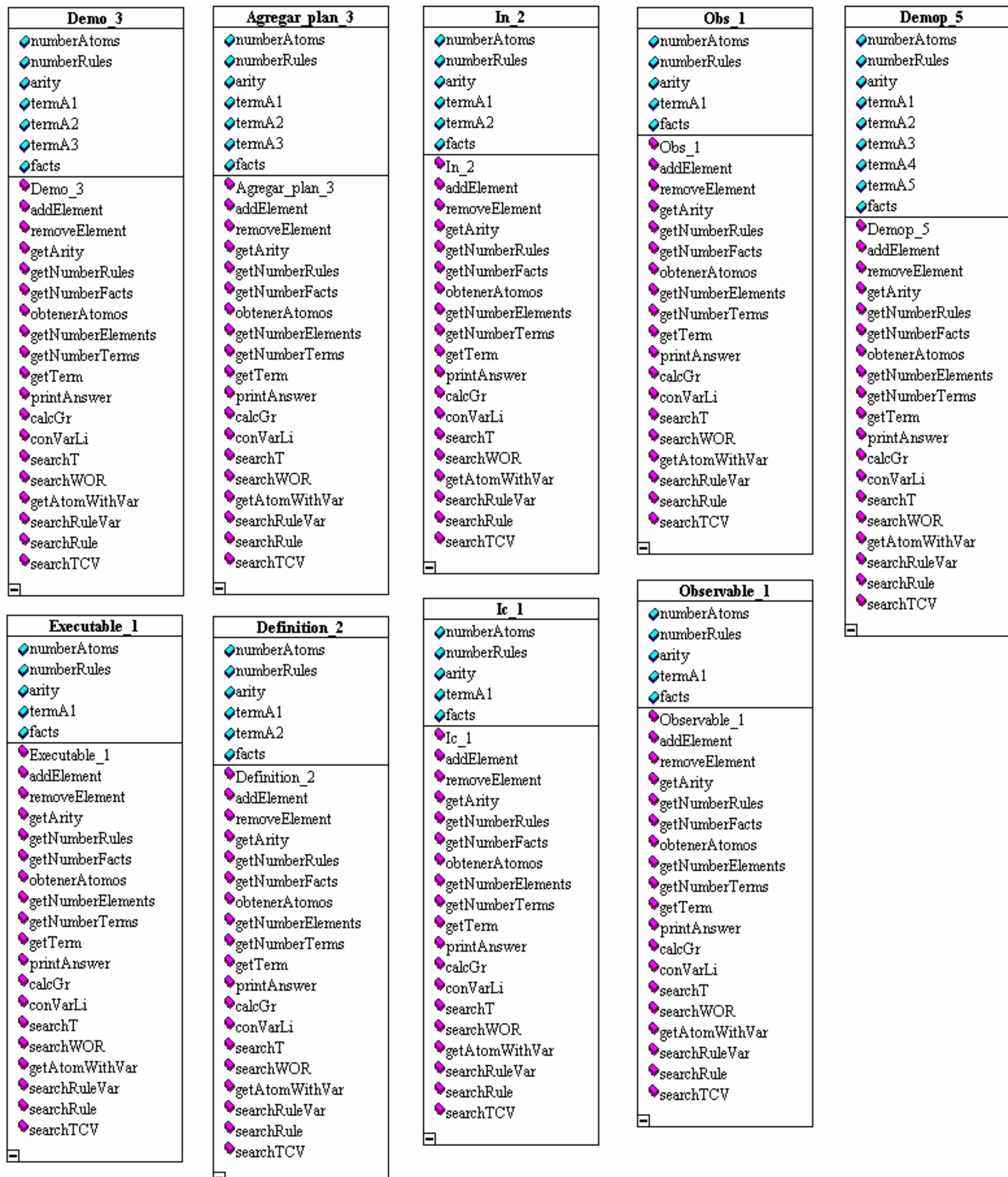

FIGURA 4.2. CLASES PARA EL MOTOR DE INFERENCIA

Al correr la consulta correspondiente se obtienen los siguientes resultados,

```
//-------------- YES  ------------
::: true
::: S/goals(splan(if(@(esta_lloviendo, true), portar_sombrilla), true), true)
::: F/[]
```

El resultado anterior se puede interpretar como sigue. Si efectivamente se corrobora que todas las observaciones son válidas, en este caso que "está lloviendo", entonces el agente debería conseguir una sombrilla.

Por otra parte, como se mencionó en el capítulo 3, la invocación del procedimiento *demo/3* permite la posibilidad de incluir metas como parámetros de entrada. Por ejemplo, podemos realizar una consulta de la siguiente forma:

```
demo(goals(splan(esta_lloviendo, splan(portar_sombrilla)), true), Meta_salida, Influencias).
```

La consulta anterior se traduciría que si se ha producido una lluvia, por lo cual se debe portar sombrilla, entonces la pregunta es qué debe hacer el agente.

La respuesta del agente, será la generación de una meta de salida, así como un conjunto de posibles planes alternativos conocidos como *Influencias*. En el caso expuesto, la respuesta será:

```
Meta_salida =  goals(splan(prestarla, true), true)
Influencias = [buscarla, asirla]
```

Lo que significa que el agente debe plantearse como meta principal el préstamo de la sombrilla. Aunque la variable asociada a las influencias arroja un plan alternativo, en caso de no poder llevar a cabo la meta de salida.

Al igual que el ejemplo ilustrado anteriormente se puede realizar modificaciones en cuanto a las consultas hechas al proyecto de Bioinformantes que se describirá en la sección siguiente, así como en cuanto a las diferentes observaciones del agente.

4.3. BIOINFORMANTES

La siguiente prueba para corroborar los mecanismos de traducción y el funcionamiento del motor de inferencia en Java, consiste en la implementación del agente en un proyecto denominado Bioinformantes planteado por Dávila et al, [DAV-2]. El objetivo del proyecto se describe a continuación.

El proyecto bioinformantes tiene como objetivo la creación de una interfaz Web, ligera e inteligente qué sirva como laboratorio virtual con las opciones proporcionadas por el programa Phylips para análisis filogenético. Éste consiste en un conjunto de programas que calculan las similitudes de secuencias genéticas y trazan los dendogramas correspondientes. La frase "agregar inteligencia a la interfaz", significa que por medio de uso de agentes de software, estos trabajen como asistentes, tutores y mineros de datos del programa Phylips [DAV-2].

La arquitectura del proyecto bioinformantes consta de cuatro componentes genéricos, a saber: servicios del sistema operativo, agentes, servidor de bioinformantes y la aplicación Web; esto en el lado del servidor. En [DAV-2] se discuten detalladamente las funciones de cada uno de los componentes. La intención del proyecto apunta a la construcción de un sistema multiagentes pero inicialmente se plantea solo la integración de herramientas existentes, unificándolas a la aplicación Web y creando el entorno virtual para los agentes inteligentes que sirvan de apoyo a sus tareas.

En nuestro caso como prueba nos interesa el componente agente, lanzado por el servidor de bioinformantes para responder alguna solicitud hecha por un usuario en la Web. El agente devolverá una respuesta al servidor de bioinformantes, que a su vez devolverá la información utilizando la plataforma Java específicamente haciendo uso de servlets.

Para verificar el funcionamiento del proyecto de Bioinformantes, tomados dos de sus versiones. En la primera se utiliza una pequena base conocimiento, similar a la de la seccion 4.2. En la segunda, se utiliza un conjunto con una mayor cantidad de predicados *definition/2, ic/2* y *executables/1.* En el anexo B, se desglosan ambas versiones.

La primera prueba consistió en la traducción de la máquina de inferencia, con las reglas asociadas a la aplicación particular, en este caso la primera versión de Bioinformantes. El proceso dió como resultado las siguientes clases:

Demop_5.java.
Agregar_plan_3.java
In_2.java.
Make_or_2.java
Aplana_2.java
Arregla_2.java
Demo_3.java
Traza_1.java
Trazac_1.java
Unfoldable_1.java
Definition_2.java
Ic_2.java
Obs_1.java
Executable_1.java
Observable_1.java

En la figura 4.3 podemos observar parte de la base de conocimiento suministrada al agente.

```
ic( sp(if(sp(biotutor_requested, sp(not(class_opened), true)), sp(open_class, true)), sp(if(sp(question_asked, true), sp(answer_question, true)), true))).

obs([biotutor_requested]).

executable(open_class).
executable(show_page_1).
executable(show_page_2).
executable(show_page_3).
executable(show_page_4).
executable(close_class).
executable(answer_question).

observable(biotutor_requested).
observable(class_opened).
observable(seen_page_1).
observable(seen_page_2).
observable(seen_page_3).
observable(seen_page_4).
observable(not_seen_anything).
observable(not_pending_queries).
observable(question_asked).
```

FIGURA 4.3. BASE DE CONOCIMIENTO PARA EL PROYECTO DE BIOINFORMANTES

Para realizar la consulta al motor de inferencia se debe definir una pregunta. En nuestro caso utilizamos una *demo(true, S, F)* – sin embargo se pueden realizar modificaciones en el formato de consulta, tal y como se discutió en la seccion anterior-, y se introduce en el programa generador de consultas, obteniendo el archivo *Query.java* con el siguiente contenido.

Luego, de lo cual se generada la consulta, se procede a ejecutarla, obteniéndose el siguiente resultado,

```
//-------------- YES  ------------
::: true
:::S/goals(sp(if(sp(biotutor_requested,sp(not(class_opened),true)),sp(open_class,            true)),
sp(if(sp(question_asked, true), sp(answer_question, true)), true)), true)
::: F/[]
```

?

El procedimiento de generación del agente para el proyecto bioinformantes a partir de su segunda versión, consistió en primer lugar en generar las clases propias de "demo". El segundo paso consistió en la definición de la base de conocimiento del agente por medio de una estructura de predicados de las definiciones, restricciones, acciones y observaciones. La base de conocimiento del agente para el proyecto de Bioinformantes se puede observar en el anexo B. Una vez realizado este paso se procedió a someter esta base de conocimiento a los mecanismos de traducción previamente discutidos y obtener la jerarquía de clases correspondientes.

Es importante resaltar en este punto, que debido a la arquitectura del agente, las restricciones y las definiciones poseen un formato particular para ser representadas. Nuestra propuesta, incluye un formato alternativo de las restricciones y definiciones, de manera tal que para el programador sea totalmente transparente. Este formato utiliza un esquema similar a la presentación de reglas de Prolog, tal como se discutió en el capítulo 3. Ahora bien, esta transparencia involucra el uso de programas diseñados para los fines descritos, como son *GenerarIC* y *GenerarDef.*

Se lleva a cabo la traducción de las definiciones y restricciones mostradas en el anexo B.2, utilizando *GenerarIC* y *GenerarDef*, dando origen a los archivos *integConst* y d*efinitions* que se muestran en las figuras 4.4 y 4.5 respectivamente. La idea subyacente, de utilizar estos programas, como se vió en la sección 3.4.2, es permitir al programador mayor flexibilidad para definir las restricciones y las definiciones.

Es importante aclarar, que nuestro diseño permite generar las clases de forma modular, es decir, una vez generados los archivos *.java* (o *.class*) bien sea de la máquina de inferencia de agente o de cualquiera de los predicados involucrados con el agente, no es necesario modificarlos todos a la vez como si formaran parte de un único archivo de Prolog. Ergo, se puede generar las clases por partes y luego colocar todos los archivos *.java* juntos como si viniesen de un único programa Prolog. Esto le permite al programador atacar el diseño del agente por etapas. Por ejemplo, puede inicialmente realizar la traducción del agente como se observó en la sección 4.2. Posteriormente podría avocarse al diseño de una serie de restricciones y definiciones adicionales o

totalmente nuevas. Luego, agruparía las clases del agente con las clases relacionadas con las restricciones y definiciones. Por último, generaría la consulta que desea realizar a la máquina de inferencia.

```
%* Conversion de las 'restricciones'
%* @author Jhon Edgar Amaya
%* @version 1.0
ic(splan(if((sesion_activa,if(tutoria_no_iniciada,if(no_metodo,if(no_datos,true)))),iniciar_tutoria
), true) ).

ic(splan(if((sesion_activa,if(no_metodo,if(tutoria_iniciada,if(mensaje_usuario_tutor_yes,if(ya_
vio_intro_uno,true))))),abrir_intro_dos), true) ).

ic(splan(if((sesion_activa,if(tutoria_iniciada,if(no_metodo,if(ya_vio_intro_dos,if(mensaje_meto
do,true))))),abrir_intro_tres_metodo), true) ).

ic(splan(if((sesion_activa,if(tutoria_iniciada,if(no_metodo,if(ya_vio_intro_dos,if(mensaje_data,
true))))),abrir_intro_tres_data), true) ).

ic(splan(if((sesion_activa,if(tutoria_iniciada,if(no_metodo,if(ya_vio_intro_tres,if(mensaje_plotti
ng,true))))),abrir_intro_cuatro_pt), true) ).

ic(splan(if((sesion_activa,if(tutoria_iniciada,if(no_metodo,if(mensaje_drawgram,if(ya_vio_intr
o_cuat_pt,true))))),abrir_intro_cinco_dg), true) ).

ic(splan(if((sesion_activa,if(tutoria_iniciada,if(metodo,if(datos,if(metodo_no_activado,if(mensa
je_usuario_tutor_yes,true)))))),iniciar_metodo), true) ).

ic(splan(if((sesion_activa,if(input_stream_lleno,true)),procesar_mensaje_input_stream),    true)
).

ic(splan(if((sesion_activa,if(mensaje_del_usuario_para_el_metodo,true)),entregar_mensaje_al
_metodo), true) ).

ic(splan(if((sesion_activa,if(mensaje_del_usuario_para_el_tutor,true)),procesar_mensaje_para
_el_tutor), true) ).

ic(splan(if((sesion_activa,if(mensaje_del_tutor_para_el_usuario,true)),enviar_mensaje_al_usu
ario), true) ).

ic(splan(if((sesion_activa,if(error_stream_lleno,true)),procesar_mensaje_error_stream), true) ).

ic(splan(if((metodo_colgado,true),recuperar_metodo), true) ).

ic(splan(if((sesion_inactiva,true),servlet_termina_tutor), true) ).

ic(splan(if((true,true),observar), true) ).
```

FIGURA 4.4. RESTRICCIONES DEL AGENTE DE BIOINFORMANTES

```
%* Conversion de las 'definitions'
%* @author Jhon Edgar Amaya
%* @version 1.0

definition(iniciar_tutoria,splan(enviar_a_usuario_intro_uno,splan(declarar_tutoria_iniciada,s
plan(declarar_ya_vio_intro_uno,splan(preguntar_usuario_si_debo_continuar,splan(recibir_re
spuesta_usuario,true)))))).

definition(abrir_intro_dos,splan(enviar_a_usuario_intro_dos,splan(declarar_ya_vio_intro_dos
,splan(solicitar_a_usuario_opcion_metodo_o_data,splan(recibir_metodo_o_data,true))))).

definition(abrir_intro_tres_metodo,splan(enviar_a_usuario_intro_tres_metodo,splan(declarar
_ya_vio_intro_tres,splan(solicitar_a_usuario_un_metodo,splan(recibir_metodo,true))))).

definition(abrir_intro_tres_data,splan(enviar_a_usuario_intro_tres_data,splan(declarar_ya_v
io_intro_tres,splan(solicitar_a_usuario_tipo_data,splan(recibir_tipo_data,true))))).

definition(abrir_intro_cuatro_pt,splan(enviar_a_usuario_intro_cuatro_pt,splan(declarar_ya_vi
o_intro_cuatro_pt,splan(solicitar_a_usuario_un_programa,splan(recibir_programa,true))))).
definition(abrir_intro_cinco_dg,splan(enviar_a_usuario_intro_cinco_dg,splan(declara_ya_vio_
intro_cinco_dg,splan(solicitar_a_usuario_permiso_para_continuar,splan(recibir_permiso_par
a_continuar,true))))).

definition(iniciar_metodo,splan(soicitar_runtime,splan(ejecutar_en_runtime_metodo,splan(de
clarar_process_id_del_metodo,splan(obtener_input_stream_del_metodo,splan(declarar_input
_stream_del_metodo,splan(obtener_output_stream_del_metodo,splan(declarar_output_strea
m_del_metodo,splan(obtener_error_stream_del_metodo,splan(declarar_error_stream_del_me
todo,splan(declarar_metodo_activo,true))))))))))).

definition(procesar_mensaje_input_stream,splan(leer_mensaje_input_stream,splan(enviar_u
suario_mensaje_metodo,splan(solicitar_respuesta_usuario,splan(recibir_respuesta,splan(de
clarar_hay_mensaje_del_usuario_para_metodo,true)))))).

definition(procesar_mensaje_para_el_tutor,splan(leer_mensaje_para_el_tutor,splan(mensaje
_es_reiniciar_metodo,splan(iniciar_metodo,true)))).

definition(procesar_mensaje_para_el_tutor,splan(leer_mensaje_para_el_tutor,splan(mensaje
_es_terminar_tutoria,splan(enviar_usuario_mensaje_despedida,splan(declarar_sesion_inacti
va,true))))).

definition(procesar_mensaje_error_stream,splan(leer_mensaje_error_stream,splan(enviar_us
uario_mensaje_error,splan(recibir_respuesta,splan(declarar_hay_mensaje_del_usuario_para
_metodo,true))))).

definition(recuperar_metodo,splan(eviar_usuario_mensaje_falla_de_metodo,splan(preguntar
_usuario_si_desea_iniciar_de_nuevo,splan(recibir_respuesta,splan(declarar_hay_mensaje_d
el_usuario_para_tutor,true))))).
```

FIGURA 4.5. DEFINICIONES DEL AGENTE DE BIOINFORMANTES

Una vez ajustado el formato de las definiciones y las constantes de restricción, procedemos a utilizar el programa de generación de clases para obtener la clase de definiciones y la asociada a las restricciones. Adicionalmente se debe generar la información asociada a los predicados *observable/1* y *executable/1*. A continuación se muestra algunos de los predicados a estos dos últimos predicados.

```
executable(leer_mensaje_para_el_tutor).
executable(enviar_a_usuario_intro_cinco_dg).
executable(enviar_a_usuario_intro_uno).
...
observable(metodo_colgado).
observable(sesion_activa).
...
```

Luego, de lo cual se genera una consulta idéntica al experimento del sección 4.3, obteniéndose el siguiente resultado,

```
//-------------- YES ------------
::: true
::: S/goals(splan(if(@(sesion_activa,if(tutoria_no_iniciada,if(no_metodo, if(no_datos, true)))), iniciar_tutoria), true),true)
::: F/[]
```

El resultado anterior se puede interpretar como sigue: Si la *sesion_activa* es cierta, -lo cual, es una condición observada por el agente mediante el predicado *observable(sesion_activa)*- ; entonces se debe *iniciar_tutoria* mediante la corroboración de las condiciones intermedias como lo son: que no se ha iniciado una sesión previamente y no existen datos ni métodos inicializados.

4.4. RESULTADOS

Con los experimentos descritos anteriormente se puede corroborar el funcionamiento efectivo tanto del traductor como de los mecanismos de búsqueda de soluciones similares a Prolog implementados en Java. A continuación, se analizarán los resultados de la propuesta en Java, y se discutirán algunos puntos importantes en cuanto al rendimiento de la misma.

4.4.1. MEDICION DEL RENDIMIENTO DE LA PROPUESTA.

Partiendo de la discusion planteada en [CLP-2][TAR-2][JIN-2], en donde se diserta sobre los requerimientos de una cantidad de tiempo particular que debe se consumido por el proceso de generación de las clases, inclusive en aquellas asociadas al motor de inferencia del agente. Ahora bien, este precio en cuanto a la generación de clases de Java, en nuestro caso es directamente proporcional a la cantidad de cláusulas, así como a la cantidad de clases existentes en el programa que se desea convertir.

Las mediciones realizadas en cuanto a la generacion de las clases en Java y su compilación respectiva, para la maquina de inferencia del agente, no supera los 75 segundos, en el peor de los casos. Pero, es importante recalcar que la mayoría de los agentes no requieren una compilación permanente o consuetudinaria. Una vez generadas las clases Java, el tiempo requerido para la búsqueda de soluciones a partir de las clases-predicados, está determinado por la estructura de las clases y la máquina virtual sobre la que se ejecutan.

Para comparar el rendimiento de la propuesta con respecto a un IDE de Prolog. Se procedió a realizar mediciones sobre el tiempo de ejecución de los programas Prolog en una plataforma nativa, en nuestro caso utilizamos el programa Amzi!. Convertimos luego los programas en Prolog, mediante el traductor de nuestra propuesta. Realizamos la medicion de tiempo de ejecución para los programas en Java.

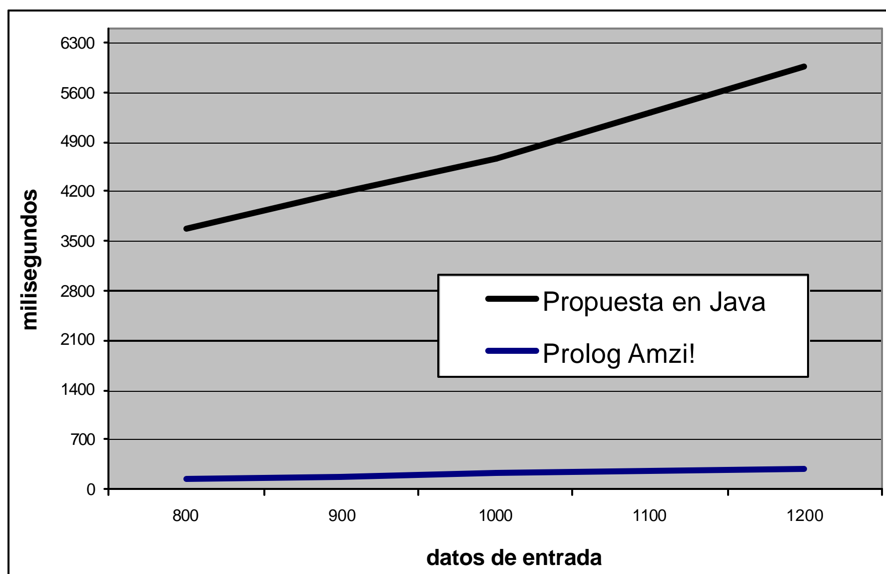

FIGURA 4.6. COMPARACIÓN ENTRE PROLOG NATIVO Y LA PROPUESTA EN JAVA BASADO EN EL TIEMPO DE CORRIDA

En la figura 4.6, se puede observar un gráfico en el que se representa el tiempo consumido en la búsqueda de soluciones, tanto por la aplicación nativa en Prolog como por la respectiva traducción en Java. Se tomó como espacio muestreal, el programa de sumatoria descrito en la sección 4.1, el cual se corrió con diferentes valores, tanto en la versión en Java como en la versión en Prolog. Adicionalmente, es pertinente mencionar que las pruebas se ejecutaron en una máquina Penthium MMX 200 MHz con 32 MB de memoria RAM.

Del gráfico se desprende algunas conclusiones importantes. Primero, que en situaciones en las cuales la profundidad de búsqueda alcanza un valor aproximado de ochocientos (800), el retardo comparativo entre ambas soluciones no supera los 3500 milisegundos, podríamos decir que este es el "precio de la multiplataformidad".

Ahora bien, es importante recalcar que en el caso de un IDE de Prolog como Amzi! u otros, su código está optimizado a través de bibliotecas nativas de la plataforma donde se ejecuta. Ergo, es de esperar que la respuesta de un programa en Prolog nativo sea mas rapida que nuestra propuesta, ya ésta es un codigo interpretado que se ejecuta sobre una máquina virtual Java.

Otra conclusión importante que se desprende del gráfico 4.6, es el hecho de la diferencia de tiempo de ejecución entre metodos anidados en nuestra propuesta, está enmarcada alrededor de 3 a 4 ms. Esta prueba nos permitió observar además, que la cantidad máxima de llamadas a métodos recursivos que puede hacer una clase Java, está alrededor de unos dos mil (2.000), lo cual representa una limitante de la implementacion en Java. En Prolog, para caso del programa de la sumatoria descrito en la sección 4.1, podría representarse con una pila de setenta y cinco mil (75.000) posiciones aproximadamente, para abarcar una profundidad de recursividad de dos mil (2.000), pero tiene la ventaja de que se puede modificar el tamaño de la pila, para incrementar la profundidad de la recursividad.

Es importante recalcar que estos datos representan resultados preliminares, se recomienda definir pruebas de rendimiento, en diferentes plataformas y con diferentes cargas de trabajo.

4.4.2. MODIFICACION DE LA CLASE DEMOP.

A la hora de utilizar el programa traductor de predicados a clases, se genera una estructura de la forma discutida en el capítulo 3, en la cual se estipula que se debe proceder a la agrupación de términos y de las reglas en dos vectores, que son a su vez los parámetros de entrada de la clase de impresión de clases. Ahora bien, en el caso del motor de inferencia, específicamente en el caso del predicado *demop/4* se tiene que al realizar el proceso de conversión de predicados, se genera una estructura que puede resumirse a través del siguiente algoritmo compuesto de sentencias secuenciales.

Verificación de todos los términos
Verificación de todas las reglas (4.4)

pero en la estructura del predicado demop/4 es:

Verificación de los dos primeros términos
(es decir, demop(G, G, [], 0) y demop(goals(true, R), R, [], _))
Verificación de todas las reglas
Verificación del ultimo término
(es decir, demop(G, G, [], _)).

Si procedemos a la ejecución del programa de la máquina de inferencia con la estructura asociada con (4.4), es decir, sin realizar ningún cambio del código arrojado por el traductor, implicará que los resultados estarían sujetos a la cláusula: *demop(G, G, [], _).*

La solución para este detallle, consiste en la separación de los términos asociados a los términos básicos en grupos diferentes, representados por dos métodos diferentes *searchTCV*. En nuestro caso, proponemos la definición del método *searchTCV2* que incluya los datos asociados a la cláusula *demop(G, G, [], _)* y la eliminación de ésta del método *searchTCV* original.

La invocación a este método se lleva a cabo a través de la inclusión del siguiente código en el método searchRule, inmediatamente después de la definición de los datos asociados a las nueve (9) reglas de *demop/4*, de la siguiente forma:

```
if (searchTCV2(X1, X2, X3, X4,CBackT) == true){
        retorno = true;
        return retorno;
}
```

Para comprobar la disertación anterior procedemos de la siguiente forma. Primero, supóngase que se tiene el escenario de pruebas de la sección 4.2. En el caso del código sin cambio generado por el traductor se obtiene:

```
//-------------- YES  ------------
::: goals(splan(esta_lloviendo,true),true)
::: S/goals(splan(if(@(esta_lloviendo,true),portar_sombrilla),splan(esta_lloviendo,true)),true)
::: F/[]
```

La salida obtenida con la propuesta de modificación del código será entonces:

```
//-------------- YES  ------------
::: goals(splan(esta_lloviendo,true),true)
::: S/goals(splan(prestarla,splan(esta_lloviendo,true)),true)
::: F/@(buscarla,@(asirla,[]))
```

Para brindar la verificación y la comprobación, se llevó el código de la máquina de inferencia a un IDE de Prolog como Amzi!, realizando la siguiente pregunta

?demo(goals(splan(esta_lloviendo,true),true), S, F). (4.5)

idéntica a la realizada en el caso referente a la comprobación a la modificación del código. Se obtiene la respuesta que corresponde a la salida de nuestro programa.

```
S = goals(splan(prestarla, splan(esta_lloviendo, true)), true)
F = [buscarla, asirla] .
yes
```

Para verificar nuestro punto de vista del problema, realizamos una estructuración del programa en Prolog similar a la mostrada en (4.4) de las cláusulas del predicado *demop/4*, y se realiza la consulta (4.5) se obteniéndose,

```
S=goals(splan(if((esta_lloviendo',' true), portar_sombrilla), splan(esta_lloviendo, true)), true)
F = '[]' .
yes
```

?

En el presente capítulo hemos descrito las pruebas desarrolladas para verificar el funcionamiento de nuestra propuesta. El conjunto de pruebas realizado, ha abarcado en primer término, aquellas pruebas en las cuales se valida el comportamiento de algunas de las características básicas de los lenguajes declarativos pero ahora en Java; se incluyen es éstas, las operaciones aritméticas, la búsqueda de soluciones a partir de un conjunto

de reglas, el manejo de la multimodalidad, entre otras. En segundo lugar, se llevo a cabo la traducción de la máquina de inferencia del agente descrito en [DAV-1] y se probó su funcionamiento con un conjunto de reglas sencillas. En tercer lugar, para verificar el alcance de la máquina de inferencia del agente se incorporó la traducción correspondiente en el proyecto Bioinformantes, tal como se describió en la sección 4.3. En última instancia se llevó a cabo, la medición del performance de nuestra propuesta, que arrojó los resultados discutidos en la sección 4.4.1.

CAPÍTULO 5. CONCLUSIONES Y RECOMENDACIONES

5.1. CONCLUSIONES

Como se demostró en el capítulo 4, los mecanismos de traducción fueron ejecutados exitosamente en el proyecto llamado Bioinformantes. Una vez, realizadas las consultas sobre las clases generadas se obtuvieron los planes correspondientes que se convierten en nuevas metas del agente. Dado que los planes son almacenados en estructuras tipo *Term,* como se discutió en el capítulo 3; y una vez implementado el agente en un proyecto como Bioinformantes, se observó la necesidad de definir una interfaz, entre las respuestas generadas por el agente y la aplicación particular que vaya a ser uso de ellas. Técnicamente consistiria en "alambrar" la salida del agente con los efectores o procesos efectores. No solamente, debe pensarse en una interfaz para las respuestas, adicionalmente deben definirse mecanismos para obtener e incorporar las nuevas observaciones a la base de conocimiento del agente, desde procesos de software o interfaces físicas. En la UNET, se está desarrollando un proyecto para definir las interfaces de físicas y lógicas para implementar el agente en un robot. En nuestra implementación utilizamos una salida similar a Prolog, y se hace uso exclusivo de la salida estándar del computador a través de la clase *Query.* El proyecto involucra la relacionar los *Terms* generados por el *Query.java* con las acciones del robot.

En el capítulo 2 se discutió la existencia de diferentes alternativas para la integración entre Prolog y Java. Nuestro objetivo como el de la mayoría de las implementaciones que combinan Prolog y Java, es la búsqueda de una arquitectura sencilla y práctica para implementar mecanismos propios de los lenguajes declarativos, como por ejemplo la resolución y la sustitución. Por esta razón, se puede observar que las clases Java presentan estructuras simples para implementar estos mecanismos. Lo anterior no significa que la simplicidad conduzca a un menoscabo de las potencialidades de estos mecanismos, todo lo contrario, ya que se ve compensado por el hecho de permitir explotar una de las características principales de Java como lo es la multiplaformidad. La simplicidad de las estructuras se puede corroborar en la implementación del control declarativo a través de los métodos *search**.

Es importante mencionar que el esquema de traducción fue realizado teniendo en cuenta una estructura dinámica como siguiente paso natural y evolutivo de nuestro proyecto. Un paso importante dado en la Tesis para la consecución del fin mencionado, lo constituye la separación de las cláusulas en dos grupos: Hechos y Reglas (En el programa se utiliza sus correspondientes nombres en inglés: facts y rules). En primer lugar, la separación permite al programador que una vez haya generado el código de las clases pueda incluir o excluir dinámicamente un hecho en la clase generada. En segundo lugar, que pueda

definirse una base de conocimiento dinámica en el caso del agente, de forma tal, que pueda extienderse las potencialidades de aplicación del agente. Para la inclusión o exclusión se hace uso de los métodos *addElement(Term)* y *removeElement(Term)* respectivamente. En la actual versión del software solo se permite la inclusión de términos *grounded*.

Como se pudo observar en el capítulo 3, se llevó a cabo la creación de un conjunto programas particulares para adecuar las estructuras vinculadas a los predicados *ic/2* y *definition/2*. La intención de este proceso es permitir a los programadores del agente definir la base de conocimiento de manera sencilla y transparente, evitando de este modo hacer uso total de la nomenclatura de la máquina de inferencia del agente y ajustarlo a una aproximación un poco más natural hacia las reglas en Prolog. Estos programas utilizan las diferentes clases definidas para el proceso de *parsing* discutidos en el capítulo 3.

La modularidad de las clases funcionales permitirá que en futuras versiones se pueda intentar implementar conceptos como CLP. La sustentación de tal afirmación, radica en que podría hacerse una modificación de las clases *Term, BscTerm, UnifTerm* y *Operators*, para incluir operaciones vinculadas con el espacio de búsqueda propio de los programas CLP, como se menciona en [CLP-1]. Adicionalmente, la estructuración de los operadores en una clase *Operators*, brinda la posibilidad de ampliar las definiciones de las operaciones aritméticas en nuevos espacios de búsqueda como en el caso de CLP.

En el capítulo 3 se describió la estructura de la clase Java que permite la asociación de predicados de Prolog, se discutió además el formato de los métodos y los atributos de la mencionada clase, inclusive se dejó entrever que una de las principales innovaciones del proyecto estriba en la inclusión de una nueva estrategia de control que denominamos "control declarativo" como parte de la clase, omitiendo de esta forma la generación de un único compilador de Prolog sobre el cual se realicen las llamadas a objetos asociados a Prolog, como por ejemplo en el caso de [INT-1] ó [JPR-1]. El control declarativo está representado por los mecanismos inherentes de los lenguajes como Prolog, entre los que destacan la resolución, la sustitución y la generación de los espacios de búsqueda. La definición del espacio de búsqueda se realiza mediante la utilización de cuatro (4) métodos específicos, que constituyen el sustento del control declarativo en las clases, ellos son: *searchT, searchRule, searchVarRule* y *searchTCV*.

Debido a las características recursivas del proceso de *parsing,* hubiese resultado más cómodo su realización del programa para la implementación de éste proceso mediante la utilización de Prolog. Si bien es cierta la premisa anterior, la implementación de los

mecanismos de *parsing* en Java, permitirá a los programadores una mayor transparencia y una mayor portabilidad.

En [DAV-1] se plantea una arquitectura de un agente que utiliza como base la programación lógica para la incorporación de características tales como, reactividad y apertura al ambiente. Para facilitar el uso de la mencionada arquitectura, en cuanto a la portabilidad, en nuestro proyecto se propone una metodología de conversión de instrucciones de Prolog a su equivalente en clases Java. Adicionalmente se creó un software que realiza de forma automática la mencionada conversión. La intención de la propuesta es crear un entorno de programación de agentes con las características presentadas en [DAV-1], de forma tal, que permita a los programadores generar una máquina de inferencia en Java a partir de las especificaciones de la mencionada arquitectura, así como, realizar modificaciones pertinentes que enriquezcan su acervo. Desde el punto de vista práctico, nuestra propuesta tuvo como norte la programación postdeclarativa [WOO-1], en la cual ciertos componentes son programados con orientación a objetos, mientras que otros lo son con lenguajes declarativos orientados a lógica.

En la propuesta se realiza una comparación continua con los mecanismos y procesos de la WAM, ya que ésta representa el punto de partida para entender la estructura de un compilador Prolog. Es importante recalcar sin embargo, que a pesar que tomamos como punto de partida la Máquina Abstracta de Warren, no es necesario que aquel programador que desee generar un traductor Prolog tenga que revisar la estructuración de la WAM inicialmente. Adicionalmente, se puede observar en nuestra propuesta la relación de algunas directivas de WAM con los métodos y atributos de la clase Java asociada a los predicados. Los apuntadores que permiten controlar la estructura principal de la WAM son sustituidos en nuestra propuesta por la estructura *while-if,* y por los métodos asociados a la búsqueda de soluciones.

Al igual como en el caso de una WAM tradicional, en la cual se utilizan los registros asociándolos a variables libres, del mismo modo en nuestra propuesta se utilizan asociaciones a objetos del tipo *Term*, los cuales cumplen una función similar a sus contrapartes en la WAM. Es importante recalcar que debido a la persistencia que poseen los objetos, específicamente el mantenimiento de los valores de los atributos de un objeto particular, se requiere la manipulación de éstos dando origen a un proceso continuo de "flush" y reinicialización, es decir, cada objeto asociado a una variable libre debe desligarse de su valor actual y fijarse en el valor que le corresponda como parámetro de inicialización cada vez que se realiza el proceso de “backtracking”.

La idea de mantener la estructura de las variables libres utilizadas en Prolog y no asociarlas directamente al tipo de dato, fue una sugerencia valiosa del Profesor Roussel, co-realizador del compilador del lenguaje Prolog, con lo cual el proceso de resolución se puede llevar a cabo con un menor costo en cuanto al tiempo de corrida, ya que no requiere la diferenciación directa del tipo de dato. Por ejemplo, en el caso de los caracteres numéricos solo se reconocen como tal al momento de llevar a cabo una operación aritmética o lógica. La asociación indirecta del tipo de dato, conllevo a la consecución de un mecanismo de optimización de la búsqueda de soluciones en el espacio de búsqueda correspondiente, de allí se desprende la reformulación del mecanismo de *while-if* planteado en el capítulo 3, en donde las operaciones aritméticas están asociadas solamente a los métodos *suma*, *resta* e *is* de la clase *Operators*.

Uno de los puntos más álgidos a la hora de estructurar la arquitectura de las clases Java asociada a los predicados, consistió en determinar cómo se llevaría a cabo la implementación y procesamiento de las estructuras denominadas términos. Los términos son la pieza fundamental sobre la que se sustentan los lenguajes declarativos. Se revisaron diferentes propuestas sobre la implementacion de téminos en Prolog, finalmente se decantó por la utilización de una modificacion de [COD-1], ya que permite mayor flexibilidad para incorporar nuevos atributos y brinda la posibilidad de extender las propiedades de los términos.

Otro punto importante en la definición de las estructuras de datos en Java, luego de haber estructurado los términos, consistió en determinar la forma de implementar las listas, que son esenciales en la programación declarativa, específicamente en Prolog. A pesar que las listas son términos per se, en la fase inicial de diseño se realizó una distinción entre un término y una lista. La razón fundamental era simplificar la fase inicial de diseño, lo cual dió origen a un abanico de posibilidades para la implementación de las listas Prolog en Java. La decisión final estuvo marcada por la necesidad de simplicidad y claridad en el momento de implementar la lista en las clases *Term y BscTerm*. Se tomó la decisión de utilizar dos esquemas independientes. El primero idéntico al descrito en el capítulo 3. El segundo, con una estructura como la planteada en [SBP-1] en la cual una lista se presenta como *[a, f, c, f],* existiendo equivalente a *.(a, .(d,.(c,.f([]))),* donde el punto ".". En nuestro caso el punto se representa a través del símbolo "@", y se asume éste último como un nombre reservado, es decir, un término especial. El segundo esquema se utiliza en la mayoría de las implementaciones de Prolog, ya que representa una manera natural y sencilla de manejar las listas.

Un punto importante discutido en [AIK-1], es la estructuración de las operaciones disyuntivas u operaciones OR, para ampliar las facilidades de programación de Prolog. En el capítulo 3 sección 4, se indicó que la invocación del procedimiento denominado "demo"

incluye un predicado de la forma: A ; B. Esta estructura corresponde a una acción A ó B, es decir, corresponde a una operación *OR* inclusiva. La existencia de este tipo de operaciones involucra una pequeña modificación del método *SearchRule*, ya que debemos introducir dentro de la estructura *while-if* la posibilidad de ejecutar la operación *OR*. Se optó por una solución que excluyese el operador "||", que representa la operación *OR* en Java, en su lugar se planteó la modificación de la estructura *while-if* por una estructura *while-if-else*.

Se ha explorado a través de la propuesta una forma para la integración de los lenguajes declarativos y orientados a objeto, específicamente para la programación de agentes inteligentes, especialmente de aquellos cuya arquitectura comprendan propiedades racionales, como en el caso [DAV-1] [KOW-1] [WOO-5]. La programación orientada a objeto, específicamente a través de Java, brinda los mecanismos necesarios para permitir una mayor portabilidad de los agentes previamente diseñados en Prolog, conduciendo de esta manera a ampliar el rango de posibilidades de aplicación del agente.

5.2. RECOMENDACIONES

En vista que el objetivo del proyecto consistía en integrar la programación declarativa e imperativa para el desarrollo de agentes, lo cual conllevo a definir una plataforma de traducción y los mecanismos de consulta y búsqueda en el espacio de búsqueda correspondiente, para verificar la funcionalidad de la integración de lenguajes. Se diseñaron clases genéricas en las cuales se implementaron las operaciones básicas que permiten realizar el procedimiento de prueba. Se propone como desarrollo posterior la incorporación de nuevas operaciones, enriqueciendo de tal forma el espectro de posibles aplicaciones del traductor. Esto último puede ser realizado modificando y agregando nuevos métodos en la clase *Operators*.

Como se discutió en las conclusiones del trabajo, el esquema de traducción fue pensado para sentarla las bases para manejar dinámicamente el codigo de los términos y reglas en Java, una vez generado el código desde Prolog. Lo anterior, tiene por objetivo disminuir el tiempo invertido en la generacion de código, mejor conocido como *tiempo de compilación*. Aumentado de esta forma el rendimiento, y extendiendo así el ambito de utilización del traductor. En una posterior mejora del software, se puede pensar en la inclusión de métodos que permita incorporar reglas de Prolog dinámicamente a la clase Java diseñada.

En el capítulo 4, se discutieron las pruebas realizadas para verificar los resultados de los mecanismos "declarativos" en Java. Ahora bien, se requiere el establecimiento de estrategias para la medición del tiempo consumido por la propuesta y compararlo con programas, ya existentes y con suficiente soporte, como por ejemplo SWI, Jinni e Interprolog. A pesar que en los resultados se plasmó una medición de tiempo de ejecución de la propuesta y un IDE de Prolog como Amzi!, se requieren pruebas más exhaustivas para compararlo con otras plataformas. En [CLP-2] se discuten algunos métodos para verificar el performance de diferentes arquitecturas de Prolog, que podrían ser útiles para definir las maquetas de pruebas de rendimiento.

Adicionalmente a las operaciones, una *"limitante"* actual del traductor automático consiste en que la pregunta que se puede realizar solo puede consistir en un único predicado. Se propone la modificación de la clase asociada a la generación de consultas para permitir múltiples preguntas simultáneamente, como por ejemplo: *pregunta1, pregunta2,..., preguntaN*; donde cada una de las preguntas consiste en un predicado; tal como se lleva a cabo en un IDE de Prolog comercial. Para llevar a cabo esta modificación, debe

realizarse una modificación de la clase *ScriptQ*, de tal forma que pueda descomponerse la clase Query en una estructura similar a:

```
 if( predicado_1.searchT(aridad, int)){
          …
          if( predicado_n.searchT(aridad, int)){
                    System.out.println("//-------------- YES  ------------");
                    System.out.println(demo.printResp(predicado_n.termA1, termino1));
                    …
           } else {
                     System.out.println("//-------------- NO  ------------");
           }
          …
} else {
          System.out.println("//-------------- NO  ------------");}
```

La palabra limitante sin embargo, no quiere significa que no puedan llevarse a cabo este tipo de consultas, solo que en la actual versión, el programador deberá incluir manualmente la generación de una cláusula de la forma,

query :- pregunta1, pregunta2,..., preguntaN.

y posteriormente, llevar a cabo la consulta sobre esta regla.

Hoy en día, la tendencia hacia software “inteligente” hace imperiosa la necesidad de investigar e implementar técnicas propias de la Inteligencia Artificial para desarrollar aplicaciones cada vez más complejas, interactivas y personalizadas. Se han difundido varias técnicas como, por ejemplo: agentes, redes neurales, lógica difusa, aprendizaje automático, entre otras. Las herramientas de software que permiten facilitar la implementacion del uso de agentes y demás técnicas mencionadas, propenderá a aumentar la calidad de las aplicaciones así como a disminuir el tiempo de desarrollo de las mismas.

Se sugiere la definición de un banco de pruebas más amplio, en las cuales se pueda medir y analizar el rendimiento de la propuesta versus otras traducciones, como por ejemplo jprolog, Interprolog, entre otras. El conjunto de pruebas, debe incluir además diferentes plataformas de hardware como diferentes sistemas operativos.

Un punto interesante a desarrollar posteriormente, incluiría la definición de una interfaz gráfica que permitiese incorporar el código del agente generado por nuestra propuesta, en una herramienta de simulación, como seria el caso de Galatea u otro programa.

ANEXOS

A. API DEL TRADUCTOR

A.1. Class Term

```
java.lang.Object
  |
  +-progtojav.Term
```

---

public class **Term**
extends java.lang.Object

Constructor Summary
Term( ) *Creates a new instance of Term*
Term(*BscTerm BscTermInit*) *Creates a new instance of Term*
Term(*java.lang.String named*)
Term(*java.lang.String named, int i*) *Creates a new instance of Term*
Term(*java.lang.String named, int i, java.lang.String h1, int i1*) *Creates a new instance of Term*
Term(*java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2*) *Creates a new instance of Term*
Term(*java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3*) *Creates a new instance of Term*
Term(*java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4*) *Creates a new instance of Term*
Term(*java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5*) *Creates a new instance of Term*
Term(*java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6*) *Creates a new instance of Term*
Term(*java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3,*

java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7)
Creates a new instance of Term

Term(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8)
Creates a new instance of Term

Term(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8, java.lang.String h9, int i9)
Creates a new instance of Term

Term(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8, java.lang.String h9, int i9, java.lang.String h10, int i10)
Creates a new instance of Term

Term(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8, java.lang.String h9, int i9, java.lang.String h10, int i10, java.lang.String h11, int i11)
Creates a new instance of Term

Term(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8, java.lang.String h9, int i9, java.lang.String h10, int i10, java.lang.String h11, int i11, java.lang.String h12, int i12)
Creates a new instance of Term

Term(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8, java.lang.String h9, int i9, java.lang.String h10, int i10, java.lang.String h11, int i11, java.lang.String h12, int i12, java.lang.String h13, int i13)
Creates a new instance of Term

Term(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8, java.lang.String h9, int i9,

java.lang.String h10, int i10, java.lang.String h11, int i11, java.lang.String h12, int i12, java.lang.String h13, int i13, java.lang.String h14, int i14)
  Creates a new instance of Term

Term(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8, java.lang.String h9, int i9, java.lang.String h10, int i10, java.lang.String h11, int i11, java.lang.String h12, int i12, java.lang.String h13, int i13, java.lang.String h14, int i14, java.lang.String h15, int i15)
  Creates a new instance of Term

Term(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8, java.lang.String h9, int i9, java.lang.String h10, int i10, java.lang.String h11, int i11, java.lang.String h12, int i12, java.lang.String h13, int i13, java.lang.String h14, int i14, java.lang.String h15, int i15, java.lang.String h16, int i16)
  Creates a new instance of Term

Term(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8, java.lang.String h9, int i9, java.lang.String h10, int i10, java.lang.String h11, int i11, java.lang.String h12, int i12, java.lang.String h13, int i13, java.lang.String h14, int i14, java.lang.String h15, int i15, java.lang.String h16, int i16, java.lang.String h17, int i17)
  Creates a new instance of Term

Term(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8, java.lang.String h9, int i9, java.lang.String h10, int i10, java.lang.String h11, int i11, java.lang.String h12, int i12, java.lang.String h13, int i13, java.lang.String h14, int i14, java.lang.String h15, int i15, java.lang.String h16, int i16, java.lang.String h17, int i17, java.lang.String h18, int i18)
  Creates a new instance of Term

Term(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8, java.lang.String h9, int i9, java.lang.String h10, int i10, java.lang.String h11, int i11, java.lang.String h12, int i12, java.lang.String h13, int i13,

java.lang.String h14, int i14, java.lang.String h15, int i15, java.lang.String h16, int i16, java.lang.String h17, int i17, java.lang.String h18, int i18, java.lang.String h19, int i19) Creates a new instance of Term
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8, java.lang.String h9, int i9, java.lang.String h10, int i10, java.lang.String h11, int i11, java.lang.String h12, int i12, java.lang.String h13, int i13, java.lang.String h14, int i14, java.lang.String h15, int i15, java.lang.String h16, int i16, java.lang.String h17, int i17, java.lang.String h18, int i18, java.lang.String h19, int i19, java.lang.String h20, int i20)* Creates a new instance of Term
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, Term j2, java.lang.String h6, int i6, java.lang.String h7, int i7, Term j3, Term j4)*
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, Term j1, Term j2, Term j3, java.lang.String h4, int i4, java.lang.String h5, int i5, java.lang.String h6, int i6, Term j4, Term j5, Term j6, Term j7, Term j8)*
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, java.lang.String h3, int i3, Term j1, Term j2, Term j3, Term j4, Term j5)*
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, Term j1, java.lang.String h3, int i3)*
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, Term j1, java.lang.String h3, int i3, java.lang.String h4, int i4)*
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, Term j1, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5)*
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, Term j1, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, Term j2, java.lang.String h6, int i6)*
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, Term j1, java.lang.String h3, int i3,*

java.lang.String h4, int i4, java.lang.String h5, int i5, Term j2, java.lang.String h6, int i6, java.lang.String h7, int i7, java.lang.String h8, int i8, Term j3, java.lang.String h9, int i9)
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, Term j1, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, Term j2, java.lang.String h6, int i6, java.lang.String h7, int i7, Term j3, java.lang.String h8, int i8, java.lang.String h9, int i9, Term j4, Term j5)*
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, Term j1, java.lang.String h3, int i3, java.lang.String h4, int i4, java.lang.String h5, int i5, Term j2, java.lang.String h6, int i6, java.lang.String h7, int i7, Term j3, Term j4)*
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, Term j1, java.lang.String h3, int i3, Term j2, java.lang.String h4, int i4)*
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, java.lang.String h2, int i2, Term j1, Term j2, Term j3, Term j4)*
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, Term j2)*
***Term**(java.lang.String named, int i, java.lang.String h1, int i1, Term j1, Term j2, Term j3)*
***Term**(java.lang.String named, int i, Term j1)*
***Term**(java.lang.String named, int i, Term j1, java.lang.String h1, int i1, java.lang.String h2, int i2, Term j2, Term j3, Term j4)*
***Term**(java.lang.String named, int i, Term j1, java.lang.String h1, int i1, Term j2, Term j3)*
***Term**(java.lang.String named, int i, Term j1, Term j2)*
***Term**(java.lang.String named, java.lang.String cond)*
***Term**(java.lang.String aux, java.lang.String named, int i, java.lang.String h1, int i1)*
***Term**(java.util.Vector BscTermnitVector)*   Creates a new instance of Term

Method Summary	
void	**addBscTermAtLast**(BscTerm BscTermino)
void	**addTerm**(BscTerm k)
void	**addTerm**(java.lang.String name, int aridad)
void	**addTermList**(Term A)
void	**asigValue**(Term t)
void	**changeTerm**(java.lang.String name, int aridad)
void	**changeTerm**(Term a)
BscTerm	**convLinkedList2Term**(java.util.LinkedList a)
java.lang.String	**elementType**()
void	**emptyTerm**()
boolean	**equals**(java.lang.String m)
boolean	**equals**(Term i)
boolean	**equals**(Term i, Term j)
Term	**functor2List**()
BscTerm	**genBscTerm**(java.util.Vector retorno2, java.lang.String nombre, int aridad)
int	**getArityTerm**()
BscTerm	**getBscTerm**(int i)
Term	**getFirstTerm**()
java.lang.String	**getFirstTermNameArity**()
java.lang.String	**getNameTerm**()
java.lang.String	**getNameTerm**(int i)
java.util.Vector	**getSubVectorBscTerm**(int i)
Term	**getTermFromBscTerm**(int valor) Convierte los subterminos BscTerm en Terminos Term
java.util.Vector	**getVectTerm**()
boolean	**isAtom**()
boolean	**isConstant**()
boolean	**isEmptyTerm**()
boolean	**isVariable**()
Term	**list2Functor**()
Term	**pieceTerm**(int i)
java.lang.String	**printListTerm**()
java.lang.String	**printTermFormProlog**()

void	***removeAddTwoTerm***(*Term A, Term B*)
void	***removeTerm***(*int i*)
int	***sizeTerm***()
void	***sustTerm***(*Term a*)
Term	***termHead***(*int i*)
Term	***termTail***(*int i*)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Term

public Term()
Creates a new instance of Term

Term

public Term(BscTerm BscTermInit)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i)
Creates a new instance of Term

Term

public Term(java.util.Vector BscTermnitVector)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3,
java.lang.String h4,
int i4)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3,
java.lang.String h4,
int i4,
java.lang.String h5,
int i5)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3,
java.lang.String h4,

int i4,
java.lang.String h5,
int i5,
java.lang.String h6,
int i6)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3,
java.lang.String h4,
int i4,
java.lang.String h5,
int i5,
java.lang.String h6,
int i6,
java.lang.String h7,
int i7)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3,
java.lang.String h4,
int i4,
java.lang.String h5,
int i5,
java.lang.String h6,
int i6,
java.lang.String h7,
int i7,
java.lang.String h8,
int i8)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,

int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3,
java.lang.String h4,
int i4,
java.lang.String h5,
int i5,
java.lang.String h6,
int i6,
java.lang.String h7,
int i7,
java.lang.String h8,
int i8,
java.lang.String h9,
int i9)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3,
java.lang.String h4,
int i4,
java.lang.String h5,
int i5,
java.lang.String h6,
int i6,
java.lang.String h7,
int i7,
java.lang.String h8,
int i8,
java.lang.String h9,
int i9,
java.lang.String h10,
int i10)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2,
java.lang.String h3,

int i3,
java.lang.String h4,
int i4,
java.lang.String h5,
int i5,
java.lang.String h6,
int i6,
java.lang.String h7,
int i7,
java.lang.String h8,
int i8,
java.lang.String h9,
int i9,
java.lang.String h10,
int i10,
java.lang.String h11,
int i11)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3,
java.lang.String h4,
int i4,
java.lang.String h5,
int i5,
java.lang.String h6,
int i6,
java.lang.String h7,
int i7,
java.lang.String h8,
int i8,
java.lang.String h9,
int i9,
java.lang.String h10,
int i10,
java.lang.String h11,
int i11,
java.lang.String h12,
int i12)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,

int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3,
java.lang.String h4,
int i4,
java.lang.String h5,
int i5,
java.lang.String h6,
int i6,
java.lang.String h7,
int i7,
java.lang.String h8,
int i8,
java.lang.String h9,
int i9,
java.lang.String h10,
int i10,
java.lang.String h11,
int i11,
java.lang.String h12,
int i12,
java.lang.String h13,
int i13)

Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3,
java.lang.String h4,
int i4,
java.lang.String h5,
int i5,
java.lang.String h6,
int i6,
java.lang.String h7,
int i7,
java.lang.String h8,
int i8,
java.lang.String h9,
int i9,
java.lang.String h10,
int i10,
java.lang.String h11,
int i11,
java.lang.String h12,

int i12,
java.lang.String h13,
int i13,
java.lang.String h14,
int i14)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3,
java.lang.String h4,
int i4,
java.lang.String h5,
int i5,
java.lang.String h6,
int i6,
java.lang.String h7,
int i7,
java.lang.String h8,
int i8,
java.lang.String h9,
int i9,
java.lang.String h10,
int i10,
java.lang.String h11,
int i11,
java.lang.String h12,
int i12,
java.lang.String h13,
int i13,
java.lang.String h14,
int i14,
java.lang.String h15,
int i15)
Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3,
java.lang.String h4,

int i4,
java.lang.String h5,
int i5,
java.lang.String h6,
int i6,
java.lang.String h7,
int i7,
java.lang.String h8,
int i8,
java.lang.String h9,
int i9,
java.lang.String h10,
int i10,
java.lang.String h11,
int i11,
java.lang.String h12,
int i12,
java.lang.String h13,
int i13,
java.lang.String h14,
int i14,
java.lang.String h15,
int i15,
java.lang.String h16,
int i16)

Creates a new instance of Term

Term

public Term(java.lang.String named,
int i,
java.lang.String h1,
int i1,
java.lang.String h2,
int i2,
java.lang.String h3,
int i3,
java.lang.String h4,
int i4,
java.lang.String h5,
int i5,
java.lang.String h6,
int i6,
java.lang.String h7,
int i7,
java.lang.String h8,
int i8,
java.lang.String h9,
int i9,
java.lang.String h10,
int i10,
java.lang.String h11,
int i11,
java.lang.String h12,

int i12,
java.lang.String h13,
int i13,
java.lang.String h14,
int i14,
java.lang.String h15,
int i15,
java.lang.String h16,
int i16,
java.lang.String h17,
int i17)

Creates a new instance of Term

Term

```
public Term(java.lang.String named,
        int i,
        java.lang.String h1,
        int i1,
        java.lang.String h2,
        int i2,
        java.lang.String h3,
        int i3,
        java.lang.String h4,
        int i4,
        java.lang.String h5,
        int i5,
        java.lang.String h6,
        int i6,
        java.lang.String h7,
        int i7,
        java.lang.String h8,
        int i8,
        java.lang.String h9,
        int i9,
        java.lang.String h10,
        int i10,
        java.lang.String h11,
        int i11,
        java.lang.String h12,
        int i12,
        java.lang.String h13,
        int i13,
        java.lang.String h14,
        int i14,
        java.lang.String h15,
        int i15,
        java.lang.String h16,
        int i16,
        java.lang.String h17,
        int i17,
        java.lang.String h18,
        int i18)
```

Creates a new instance of Term

Term

```
public Term(java.lang.String named,
            int i,
            java.lang.String h1,
            int i1,
            java.lang.String h2,
            int i2,
            java.lang.String h3,
            int i3,
            java.lang.String h4,
            int i4,
            java.lang.String h5,
            int i5,
            java.lang.String h6,
            int i6,
            java.lang.String h7,
            int i7,
            java.lang.String h8,
            int i8,
            java.lang.String h9,
            int i9,
            java.lang.String h10,
            int i10,
            java.lang.String h11,
            int i11,
            java.lang.String h12,
            int i12,
            java.lang.String h13,
            int i13,
            java.lang.String h14,
            int i14,
            java.lang.String h15,
            int i15,
            java.lang.String h16,
            int i16,
            java.lang.String h17,
            int i17,
            java.lang.String h18,
            int i18,
            java.lang.String h19,
            int i19)
```

Creates a new instance of Term

Term

```
public Term(java.lang.String named,
            int i,
            java.lang.String h1,
            int i1,
            java.lang.String h2,
            int i2,
            java.lang.String h3,
```

int i3,
java.lang.String h4,
int i4,
java.lang.String h5,
int i5,
java.lang.String h6,
int i6,
java.lang.String h7,
int i7,
java.lang.String h8,
int i8,
java.lang.String h9,
int i9,
java.lang.String h10,
int i10,
java.lang.String h11,
int i11,
java.lang.String h12,
int i12,
java.lang.String h13,
int i13,
java.lang.String h14,
int i14,
java.lang.String h15,
int i15,
java.lang.String h16,
int i16,
java.lang.String h17,
int i17,
java.lang.String h18,
int i18,
java.lang.String h19,
int i19,
java.lang.String h20,
int i20)

Creates a new instance of Term

Term

public Term(java.lang.String aux,
java.lang.String named,
int i,
java.lang.String h1,
int i1)

Term

public Term(java.lang.String named,
int i,
Term j1,
Term j2)

Term

```
public Term(java.lang.String named,
            int i,
            Term j1)
```

Term

```
public Term(java.lang.String named,
            int i,
            java.lang.String h1,
            int i1,
            Term j1,
            Term j2,
            Term j3)
```

Term

```
public Term(java.lang.String named,
            int i,
            java.lang.String h1,
            int i1,
            java.lang.String h2,
            int i2,
            Term j1,
            Term j2,
            Term j3,
            Term j4)
```

Term

```
public Term(java.lang.String named,
            int i,
            Term j1,
            java.lang.String h1,
            int i1,
            Term j2,
            Term j3)
```

Term

```
public Term(java.lang.String named,
            int i,
            java.lang.String h1,
            int i1,
            Term j2)
```

Term

```
public Term(java.lang.String named,
            int i,
            java.lang.String h1,
            int i1,
            java.lang.String h2,
            int i2,
            java.lang.String h3,
```

```
        int i3,
        Term j1,
        Term j2,
        Term j3,
        Term j4,
        Term j5)
```

Term

```
public Term(java.lang.String named,
        int i,
        java.lang.String h1,
        int i1,
        java.lang.String h2,
        int i2,
        Term j1,
        java.lang.String h3,
        int i3,
        java.lang.String h4,
        int i4,
        java.lang.String h5,
        int i5)
```

Term

```
public Term(java.lang.String named,
        int i,
        java.lang.String h1,
        int i1,
        java.lang.String h2,
        int i2,
        Term j1,
        java.lang.String h3,
        int i3)
```

Term

```
public Term(java.lang.String named,
        int i,
        java.lang.String h1,
        int i1,
        java.lang.String h2,
        int i2,
        Term j1,
        java.lang.String h3,
        int i3,
        java.lang.String h4,
        int i4)
```

Term

```
public Term(java.lang.String named,
        int i,
        java.lang.String h1,
```

```
        int i1,
        java.lang.String h2,
        int i2,
        Term j1,
        java.lang.String h3,
        int i3,
        Term j2,
        java.lang.String h4,
        int i4)
```

Term

```
public Term(java.lang.String named,
        int i,
        java.lang.String h1,
        int i1,
        java.lang.String h2,
        int i2,
        Term j1,
        java.lang.String h3,
        int i3,
        java.lang.String h4,
        int i4,
        java.lang.String h5,
        int i5,
        Term j2,
        java.lang.String h6,
        int i6,
        java.lang.String h7,
        int i7,
        java.lang.String h8,
        int i8,
        Term j3,
        java.lang.String h9,
        int i9)
```

Term

```
public Term(java.lang.String named,
        int i,
        java.lang.String h1,
        int i1,
        java.lang.String h2,
        int i2,
        Term j1,
        java.lang.String h3,
        int i3,
        java.lang.String h4,
        int i4,
        java.lang.String h5,
        int i5,
        Term j2,
        java.lang.String h6,
        int i6)
```

Term

```
public Term(java.lang.String named,
            int i,
            java.lang.String h1,
            int i1,
            java.lang.String h2,
            int i2,
            java.lang.String h3,
            int i3,
            Term j2,
            java.lang.String h6,
            int i6,
            java.lang.String h7,
            int i7,
            Term j3,
            Term j4)
```

Term

```
public Term(java.lang.String named,
            int i,
            java.lang.String h1,
            int i1,
            java.lang.String h2,
            int i2,
            Term j1,
            java.lang.String h3,
            int i3,
            java.lang.String h4,
            int i4,
            java.lang.String h5,
            int i5,
            Term j2,
            java.lang.String h6,
            int i6,
            java.lang.String h7,
            int i7,
            Term j3,
            Term j4)
```

Term

```
public Term(java.lang.String named,
            int i,
            java.lang.String h1,
            int i1,
            java.lang.String h2,
            int i2,
            Term j1,
            java.lang.String h3,
            int i3,
            java.lang.String h4,
```

```
        int i4,
        java.lang.String h5,
        int i5,
        Term j2,
        java.lang.String h6,
        int i6,
        java.lang.String h7,
        int i7,
        Term j3,
        java.lang.String h8,
        int i8,
        java.lang.String h9,
        int i9,
        Term j4,
        Term j5)
```

Term

```
public Term(java.lang.String named,
        int i,
        Term j1,
        java.lang.String h1,
        int i1,
        java.lang.String h2,
        int i2,
        Term j2,
        Term j3,
        Term j4)
```

Term

```
public Term(java.lang.String named,
        int i,
        java.lang.String h1,
        int i1,
        java.lang.String h2,
        int i2,
        java.lang.String h3,
        int i3,
        Term j1,
        Term j2,
        Term j3,
        java.lang.String h4,
        int i4,
        java.lang.String h5,
        int i5,
        java.lang.String h6,
        int i6,
        Term j4,
        Term j5,
        Term j6,
        Term j7,
        Term j8)
```

Term

public Term(java.lang.String named,
java.lang.String cond)

Term

public Term(java.lang.String named)

Method Detail

getNameTerm

public java.lang.String getNameTerm()

---

getNameTerm

public java.lang.String getNameTerm(int i)

---

getFirstTerm

public Term getFirstTerm()

---

getFirstTermNameArity

public java.lang.String getFirstTermNameArity()

---

getArityTerm

public int getArityTerm()

---

addBscTermAtLast

public void addBscTermAtLast(BscTerm BscTermino)

---

sizeTerm

public int sizeTerm()

---

isAtom

public boolean isAtom()

---

isEmptyTerm

public boolean isEmptyTerm()

---

getBscTerm

public BscTerm getBscTerm(int i)

---

getVectTerm

public java.util.Vector getVectTerm()

---

getSubVectorBscTerm

public java.util.Vector getSubVectorBscTerm(int i)

---

isConstant

public boolean isConstant()

---

isVariable

public boolean isVariable()

---

addTermList

public void addTermList(Term A)

---

addTerm

public void addTerm(java.lang.String name,
int aridad)

---

addTerm

public void addTerm(BscTerm k)

---

sustTerm

public void sustTerm(Term a)

---

changeTerm

public void changeTerm(java.lang.String name,
int aridad)

---

changeTerm

public void changeTerm(Term a)

---

removeTerm

public void removeTerm(int i)

---

removeAddTwoTerm

public void removeAddTwoTerm(Term A,
Term B)

---

asigValue

public void asigValue(Term t)

---

elementType

public java.lang.String elementType()

getTermFromBscTerm

public Term getTermFromBscTerm(int valor)
Convierte un los subterminos BscTerm en Terminos Term Ej.

---

emptyTerm

public void emptyTerm()

---

list2Functor

public Term list2Functor()

---

functor2List

public Term functor2List()

---

printTermFormProlog

public java.lang.String printTermFormProlog()

---

printListTerm

public java.lang.String printListTerm()

---

termHead

public Term termHead(int i)

---

termTail

public Term termTail(int i)

---

convLinkedList2Term

public BscTerm convLinkedList2Term(java.util.LinkedList a)

---

genBscTerm

public BscTerm genBscTerm(java.util.Vector retorno2,
java.lang.String nombre,
int aridad)

---

pieceTerm

public Term pieceTerm(int i)

---

equals

public boolean equals(Term i,
Term j)

---

equals

public boolean equals(java.lang.String m)

---

equals

public boolean equals(Term i)

A.2. Class Operators

java.lang.Object
  |
  *+-**progtojav.Operators***

---

*public class **Operators***
extends java.lang.Object

Constructor Summary
Operators( ) *Creates a new instance of Operators*

Method Summary	
void	**addOper**(java.lang.String Oper, java.lang.String Prio)
void	**addOperCl**(java.lang.String simbolo, int valor)
void	**addOperCl**(java.lang.String simbolo, java.lang.String simboloSustitucion, int valor) *Metodos para realizar el ordenamiento de los valores de precedencia*
boolean	**assertl**(Term a)
boolean	**atom**(Term A) *OPERACION ATOM*
boolean	**consOp**(java.lang.String op)
java.lang.String	**convert**(java.lang.String stringName)
java.lang.String	**convert2**(java.lang.String stringWork)
java.lang.String	**convertJava**(java.lang.String nameClass)
boolean	**faill**( ) *OPERACION FAIL*
boolean	**haltl**( )
boolean	**ident**(Term X, Term Y) *OPERACION IDENT*
boolean	**is**(java.lang.String X, int Y)
boolean	**is**(java.lang.String X, int Y, int Z, java.lang.String op) *OPERACION IS*

boolean	**is**(Term X, Term Y)
Term	**ist**(Term X, Term Y)
boolean	**mayor**(Term X, Term Y) OPERACION MAYOR
boolean	**menor**(Term X, Term Y) OPERACION MENOR
boolean	**newl**() OPERACION NEW
boolean	**nident**(Term X, Term Y) OPERACION NIDENT
boolean	**not**(boolean cond) OPERACION NOT
boolean	**nunif**(Term X, Term Y) OPERACION NUNIF
Term	**nunift**(Term X, Term Y)
boolean	**or**(Term X, Term Y) OPERACION OR
void	**orden**()
java.lang.String	**posEnd**(int i)
java.lang.String	**posSta**(int i)
Term	**resta**(Term X, Term Y) OPERACION RESTA
boolean	**retractl**(Term a) Metodos experimentales
int	**sizeVecOper**()
Term	**suma**(Term X, Term Y) OPERACION SUMA
boolean	**tabl**(Term X) OPERACION TAB
boolean	**unif**(Term X, Term Y) OPERACION UNIF
Term	**unift**(Term X, Term Y)
boolean	**univ**(Term X, Term Y) OPERACION UNIV
boolean	**writel**(Term A) OPERACION WRITE

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Operators

public ***Operators****()*
Creates a new instance of Operators

Method Detail

not

public boolean ***not****(boolean cond)*
OPERACION NOT

---

is

public boolean ***is****(java.lang.String X,*
int Y,
int Z,
java.lang.String op)
OPERACION IS

---

is

public boolean ***is****(java.lang.String X,*
int Y)

---

is

public boolean ***is****(Term X,*
Term Y)

---

ist

public Term ***ist****(Term X,*
Term Y)

---

suma

public Term ***suma****(Term X,*
Term Y)
OPERACION SUMA

---

resta

public Term ***resta****(Term X,*
Term Y)
OPERACION RESTA

---

menor

public boolean ***menor****(Term X,*
Term Y)
OPERACION MENOR

---

mayor

public boolean ***mayor****(Term X,*

Term Y)
OPERACION MAYOR

---

unif

public boolean ***unif****(Term X,*
Term Y)
OPERACION UNIF

---

unift

public Term ***unift****(Term X,*
Term Y)

---

nunif

public boolean ***nunif****(Term X,*
Term Y)
OPERACION NUNIF

---

nunift

public Term ***nunift****(Term X,*
Term Y)

---

ident

public boolean ***ident****(Term X,*
Term Y)
OPERACION IDENT

---

nident

public boolean ***nident****(Term X,*
Term Y)
OPERACION NIDENT

---

or

public boolean ***or****(Term X,*
Term Y)
OPERACION OR

---

univ

public boolean ***univ****(Term X,*
Term Y)
OPERACION UNIV

---

atom

public boolean ***atom****(Term A)*
OPERACION ATOM

---

faill

*public boolean **faill**()*
OPERACION FAIL

---

newl

*public boolean **newl**()*
OPERACION NEW

---

tabl

*public boolean **tabl**(Term X)*
OPERACION TAB

---

writel

*public boolean **writel**(Term A)*
OPERACION WRITE

---

retractl

*public boolean **retractl**(Term a)*
Metodos experimentales

---

assertl

*public boolean **assertl**(Term a)*

---

convertJava

*public java.lang.String **convertJava**(java.lang.String nameClass)*

---

haltl

*public boolean **haltl**()*

---

addOperCl

*public void **addOperCl**(java.lang.String simbolo,*
java.lang.String simboloSustitucion,
int valor)
Metodos para realizar la ordenamiento de los valores de precedencia

---

addOperCl

*public void **addOperCl**(java.lang.String simbolo,*
int valor)

---

addOper

*public void **addOper**(java.lang.String Oper,*
java.lang.String Prio)

---

consOp

*public boolean **consOp**(java.lang.String op)*

---

orden

public void ***orden****()*

---

sizeVecOper

public int ***sizeVecOper****()*

---

posSta

public java.lang.String ***posSta****(int i)*

---

posEnd

public java.lang.String ***posEnd****(int i)*

---

convert

public java.lang.String ***convert****(java.lang.String stringName)*

---

convert2

public java.lang.String ***convert2****(java.lang.String stringWork)*

A.3. Class BscTerm

java.lang.Object
  |
  *+-**progtojav.BscTerm***

---

public class ***BscTerm***
extends java.lang.Object

Constructor Summary
BscTerm( )   *Creates a new instance of BscTerm*
BscTerm(java.lang.String name, int arity)   *Creates a new instance of BscTerm*
BscTerm(java.lang.String name, int arity, BscTerm Term)   *Creates a new instance of BscTerm*
BscTerm(java.lang.String name, int arity, BscTerm Term1, BscTerm Term2)   *Creates a new instance of BscTerm*
BscTerm(java.lang.String name, int arity, BscTerm Term1, BscTerm Term2, BscTerm Term3)   *Creates a new instance of BscTerm*
BscTerm(java.lang.String name, int arity, BscTerm Term1, BscTerm Term2, BscTerm Term3, BscTerm Term4)   *Creates a new instance of BscTerm*
BscTerm(java.lang.String name, int arity, BscTerm Term1, BscTerm Term2, BscTerm Term3, BscTerm Term4, BscTerm Term5)   *Creates a new instance of BscTerm*

BscTerm(java.lang.String name, int arity, java.util.Vector subTerms) Creates a new instance of BscTerm

Method Summary	
void	**addSubTerm**(BscTerm term)
java.util.LinkedList	**descomp**(java.util.LinkedList list)
boolean	**equal**(BscTerm Term)
boolean	**equal**(java.lang.String stringWork)
java.util.Vector	**getAllSubTerms**()
int	**getArity**()
java.lang.String	**getName**()
BscTerm	**getSubTerm**(int i)
boolean	**isAtom**()
java.lang.String	**isType**()
java.lang.String	**printBscTerm**()
java.lang.String	**printTermFormProlog**()
void	**setArity**(int arity)
void	**setName**(java.lang.String name)
int	**sizeSubTerms**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

BscTerm

public **BscTerm**()
Creates a new instance of BscTerm

BscTerm

public **BscTerm**(java.lang.String name,
int arity)
Creates a new instance of BscTerm

BscTerm

public **BscTerm**(java.lang.String name,
int arity,
java.util.Vector subTerms)
Creates a new instance of BscTerm

BscTerm

public ***BscTerm***(java.lang.String name,
int arity,
BscTerm Term)
Creates a new instance of BscTerm

BscTerm

public ***BscTerm***(java.lang.String name,
int arity,
BscTerm Term1,
BscTerm Term2)
Creates a new instance of BscTerm

BscTerm

public ***BscTerm***(java.lang.String name,
int arity,
BscTerm Term1,
BscTerm Term2,
BscTerm Term3)
Creates a new instance of BscTerm

BscTerm

public ***BscTerm***(java.lang.String name,
int arity,
BscTerm Term1,
BscTerm Term2,
BscTerm Term3,
BscTerm Term4)
Creates a new instance of BscTerm

BscTerm

public ***BscTerm***(java.lang.String name,
int arity,
BscTerm Term1,
BscTerm Term2,
BscTerm Term3,
BscTerm Term4,
BscTerm Term5)
Creates a new instance of BscTerm

Method Detail

getName

public java.lang.String ***getName***()

---

getArity

public int ***getArity***()

---

getAllSubTerms

public java.util.Vector ***getAllSubTerms***()

---

getSubTerm

*public BscTerm **getSubTerm**(int i)*

---

sizeSubTerms

*public int **sizeSubTerms**()*

---

setName

*public void **setName**(java.lang.String name)*

---

setArity

*public void **setArity**(int arity)*

---

addSubTerm

*public void **addSubTerm**(BscTerm term)*

---

isAtom

*public boolean **isAtom**()*

---

isType

*public java.lang.String **isType**()*

---

printTermFormProlog

*public java.lang.String **printTermFormProlog**()*

---

printBscTerm

*public java.lang.String **printBscTerm**()*

---

descomp

*public java.util.LinkedList **descomp**(java.util.LinkedList list)*

---

equal

*public boolean **equal**(java.lang.String stringWork)*

---

equal

*public boolean **equal**(BscTerm Term)*

A.4. Class ParseTerm

java.lang.Object
  |
  *+-**progtojav.ParseTerm***

*public class **ParseTerm***
extends java.lang.Object

Constructor Summary
***ParseTerm**( )* *Creates a new instance of parseTerm*

Method Summary	
void	***addParseSubTerm**(ParseTerm Term)*
void	***arityPlusOne**( )*
BscTerm	***convertToBscTerm**( )*
ParseTerm	***copyParseTerm**( )*
int	***getArity**( )*
java.lang.String	***getName**( )*
ParseTerm	***getParseSubTerm**(int i)*
java.lang.String	***getParseTermSpec**( )*
int	***getSizeParseSubTerm**( )*
java.lang.String	***getTermProlog**( )*
java.util.Vector	***getVectorParseSubTerm**( )*
void	***initParseTerm**(java.lang.String name, int arity)*
void	***initParseTerm**(java.lang.String name, int arity, java.util.Vector subTerm)*
boolean	***isEnd**( )*
boolean	***isFunctor**( )*
boolean	***isParseSubTermEmpty**( )*
boolean	***isVariable**( )*
void	***nullParseTerm**( )*
void	***printAllParseTerm**( )*
java.lang.String	***printParseTerm**( )*
void	***removeParseSubTerm**(int i)*
void	***removeSubTermFirst**( )*

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

ParseTerm

*public **ParseTerm**()*
Creates a new instance of parseTerm

Method Detail

initParseTerm

*public void **initParseTerm**(java.lang.String name,*
int arity,
java.util.Vector subTerm)

---

initParseTerm

*public void **initParseTerm**(java.lang.String name,*
int arity)

---

getName

*public java.lang.String **getName**()*

---

getArity

*public int **getArity**()*

---

getParseSubTerm

*public ParseTerm **getParseSubTerm**(int i)*

---

getVectorParseSubTerm

*public java.util.Vector **getVectorParseSubTerm**()*

---

nullParseTerm

*public void **nullParseTerm**()*

---

isVariable

*public boolean **isVariable**()*

---

isFunctor

*public boolean **isFunctor**()*

---

isParseSubTermEmpty

*public boolean **isParseSubTermEmpty**()*

---

addParseSubTerm

public void ***addParseSubTerm****(ParseTerm Term)*

---

removeParseSubTerm

public void ***removeParseSubTerm****(int i)*

---

removeSubTermFirst

public void ***removeSubTermFirst****()*

---

getSizeParseSubTerm

public int ***getSizeParseSubTerm****()*

---

arityPlusOne

public void ***arityPlusOne****()*

---

printParseTerm

public java.lang.String ***printParseTerm****()*

---

isEnd

public boolean ***isEnd****()*

---

printAllParseTerm

public void ***printAllParseTerm****()*

---

getParseTermSpec

public java.lang.String ***getParseTermSpec****()*

---

copyParseTerm

public ParseTerm ***copyParseTerm****()*

---

getTermProlog

public java.lang.String ***getTermProlog****()*

---

convertToBscTerm

public BscTerm ***convertToBscTerm****()*

B. REGLAS DEL AGENTE DE BIOINFORMANTES.

B.1. Versión 1.0

```
% biobraint.pl
% A version of the iff proof procedure of GLORIA for proposicional logic programs with negation
% This version has been reduced to eliminate :- op and so defined operators.
% Author:  Jacinto Davila
% Date: 3/13/2002
% Date: 4/23/2002

% Stop reasoning when there are no more resources
demop( _, G, G, [], 0 ).

% Stop reasoning when a plan is completed.
demop( _, goals(true, R), R, [], _).
% cut a failing plan
demop( Obs, goals( sp( false, _), Others), G, I, R ) :-
   demop( Obs, Others, G, I, R ).

% Negation
demop( Obs, goals( sp( not(A), RP ), RG ), NGoals, Influences, R ) :-
   atom(A),
   NR is R - 1,
   traza(['transform negation into ic ', A, ' -> false ']),
   demop( Obs, goals( sp( if( sp(A, true), false), RP), RG ), NGoals, Influences, NR ).

% Propagation.
demop( Obs, goals( sp( if( sp(A, B), C ), RP ), RG ), NGoals, Influences, R ) :-
   atom(A),
   (in(A, RP); member(A,Obs)),
   NR is R - 1,
   traza(['propagates ', A]),
   demop( Obs, goals( sp( if( B, C), RP), RG ), NGoals, Influences, NR ).

% Unfolding.
demop( Obs, goals( sp( if( sp(A, B), C ), RP ), RG ), NGoals, Influences, R ) :-
   atom(A),
   unfoldable(A),
   definition(A, Def ),
   NR is R - 1,
   traza(['unfold ', A, ' into ', Def]),
   demop( Obs, goals( sp( if( sp(Def, B), C), RP), RG ), NGoals, Influences, NR ).

% Negation in the body of an implication
demop( Obs, goals( sp( if( sp( not(A), B), C ), RP ), RG ), NGoals, Influences, R ) :-
   atom(A),
   NR is R - 1,
   traza(['deals with not ', A, ' in the body of an implication' ]),
   demop(Obs,goals(sp(if(B,sp(or(sp(A,true),or(C, false)),true)),RP),RG),NGoals,Influences,NR ).
```

```
% true -> C  is equivalent to C
demop( Obs, goals( sp( if( true, C ), RP ), RG ), NGoals, Influences, R ) :-
   NR is R - 1,
   traza( ['cleans true body of ', C] ),
   agregar_plan(C, RP, NP),
   demop( Obs, goals( NP, RG ), NGoals, Influences, NR ).

% ( A or B ) -> C is equivalent to A -> C and B -> C
demop( Obs, goals( sp( if( sp( or(A, B), C), D ), RP ), RG ),NGoals,Influences, R ):-
   NR is R - 1,
   traza(['distributes if ', A, ' or ', B ,' then ', C]),
   demop(Obs, goals( sp( if( sp(A, C), D), sp( if( (B, C), D), RP ) ), RG ), NGoals, Influences, NR ).

% (A or B) and C is equivalent to A and C or B and C
demop( Obs, goals( sp( or(A,  B), RP ), RG ), NGoals, Influences, R ) :-
   agregar_plan( A, RP, NA), % agregar_plan( B, RP, NB),
   NR is R - 1,
  traza(['distributes ', A, ' or ', B, ' over ', RP ]),
  demop( Obs, goals( NA, goals( sp(B, RP), RG ) ), NGoals, Influences, NR ).
% Simplification: A and A is equivalent to A
demop( Obs, goals( sp( A, RP ), RG ), NGoals, Influences, R ) :-
   atom(A),
   in(A, RP),
   NR is R - 1,
   traza(['simplifies ', A, RP ]),
   demop( Obs, goals( RP, RG ), NGoals, Influences, NR ).

% Unfolding: A <- Def, A and RP is equivalent to Def and RP
demop( Obs, goals( sp( A, RP ), RG ), NGoals, Influences, R ) :-
   atom(A),
   unfoldable(A),
   definition(A, Def),
   NR is R - 1,
   traza(['unfolds ', A, Def]),
   demop( Obs, goals( sp( Def, RP), RG ), NGoals, Influences, NR ).

% Abduction to produce influences..
demop( Obs, goals( sp( A, RP ), RG ), NGoals, [A | Influences], R ) :-
   atom(A),
   executable(A),
   traza(['abduces ', A]),
   NR is R - 1,
   demop( Obs, goals( RP, RG ), NGoals, Influences, NR ).

% Abduction to consume observations.
demop( Obs, goals( sp( A, RP ), RG ), NGoals, Influences, R ) :-
   atom(A),
   observable(A),
   member(A, Obs),
   NR is R - 1,
   traza(['consumes ', A]),
   demop( Obs, goals( RP, RG ), NGoals, Influences, NR ).

% Removal of plans for not observing on time.
```

```
demop( Obs, goals( sp( A, RP ), RG ), NGoals, Influences, R ) :-
   atom(A),
   observable(A),
   not(member(A, Obs)),
   NR is R - 1,
   traza(['prunes ', A, RP]),
   demop( Obs, RG, NGoals, Influences, NR ).

% Can't to anything..
demop(_, G, G, [], _).

% Add plans with the structure sp(First Action, Rest of Plan)
agregar_plan( true, X, X ).
agregar_plan( sp( A, X), Y, sp(A, Z) ) :- agregar_plan(X, Y, Z).

in(A, sp(A, _)).
in(A, sp(_,R)) :- in(A, R).

% Transforma  [] list into a (.. , false) or-list
make_or([], false).
make_or([A|B], or(AA,BB)) :- arregla(A,AA), make_or(B, BB).

aplana([], true).
aplana([C|R], sp(C, RR)) :- aplana(R, RR).

arregla(A, sp(A,true)) :- atom(A) ; A = not(_).
arregla((A,B), sp(A,BB)) :- arregla(B,BB).

% Invoking demo
demo(Gin, Gout, Influences ) :-  ic( IC ), obs( Obs ),
  ( Gin = goals(Plan, Rest) ; ( Plan = true, Rest = true ) ),
  agregar_plan( IC, Plan, NPlan ), % IC are put first into the plan.
  demop( Obs, goals(NPlan, Rest),  Gout, Influences, 200 ).

% test
c(I, A) :-
    demo( true, A, I ).

traza([]) :- nl, !.
traza([A|R]) :- atom(A), write(A), !, nl, traza(R).
traza([A|R]) :- tab(3), trazac(A), traza(R).

trazac(sp(A,R)) :- write(A), write(','), write(R), !.
trazac(goals(A,R)) :- write('Goals '), write(A), write(','), write(R), !.
trazac(R) :- write(R).

unfoldable(A) :- not(executable(A)), not(observable(A)).

definition(breath, or( sp(do_nothing, true), false) ).

ic( sp(if(sp(biotutor_requested, sp(not(class_opened), true)), sp(open_class, true)),
sp(if(sp(question_asked, true), sp(answer_question, true)), true))).

obs([biotutor_requested]).
```

executable(open_class).
executable(show_page_1).
executable(show_page_2).
executable(show_page_3).
executable(show_page_4).
executable(close_class).
executable(answer_question).

observable(biotutor_requested).
observable(class_opened).
observable(seen_page_1).
observable(seen_page_2).
observable(seen_page_3).
observable(seen_page_4).
observable(not_seen_anything).
observable(not_pending_queries).
observable(question_asked).

B.2. Versión 2.0

% --------------------------- Restricciones ---------------------------------------

if sesion_activa, tutoria_no_iniciada, no_metodo, no_datos then iniciar_tutoria.

if sesion_activa, no_metodo, tutoria_iniciada, mensaje_usuario_tutor_yes, ya_vio_intro_uno then abrir_intro_dos.

if sesion_activa, tutoria_iniciada, no_metodo, ya_vio_intro_dos, mensaje_metodo then abrir_intro_tres_metodo.

if sesion_activa, tutoria_iniciada, no_metodo, ya_vio_intro_dos, mensaje_data then abrir_intro_tres_data.

if sesion_activa, tutoria_iniciada, no_metodo, ya_vio_intro_tres, mensaje_plotting then abrir_intro_cuatro_pt.

if sesion_activa, tutoria_iniciada, no_metodo, mensaje_drawgram, ya_vio_intro_cuat_pt then abrir_intro_cinco_dg.

if sesion_activa, tutoria_iniciada, metodo, datos, metodo_no_activado, mensaje_usuario_tutor_yes then iniciar_metodo.

if sesion_activa, input_stream_lleno then procesar_mensaje_input_stream.

if sesion_activa, mensaje_del_usuario_para_el_metodo then entregar_mensaje_al_metodo.

if sesion_activa, mensaje_del_usuario_para_el_tutor then procesar_mensaje_para_el_tutor.

```
if sesion_activa, mensaje_del_tutor_para_el_usuario then enviar_mensaje_al_usuario.

if sesion_activa, error_stream_lleno then procesar_mensaje_error_stream.

if metodo_colgado then recuperar_metodo.

if sesion_inactiva then servlet_termina_tutor.

% tambien se ponen en JAVA todas las que tengan que ver con observar.. solo dejamos
% una que dispare TODAS las actualizaciones.

if true then observar.

% --------------- Definiciones -----------------------------------

iniciar_tutoria :-
        enviar_a_usuario_intro_uno,
        declarar_tutoria_iniciada,
        declarar_ya_vio_intro_uno,
        preguntar_usuario_si_debo_continuar,
           recibir_respuesta_usuario.
        suspender_sesion.

% en el caso de no poder iniciar el tutor NO debemos tener una regla.
% simplemente no hace nada nuestro agente...

abrir_intro_dos :-
        enviar_a_usuario_intro_dos,
        declarar_ya_vio_intro_dos,
        solicitar_a_usuario_opcion_metodo_o_data,
        recibir_metodo_o_data.

abrir_intro_tres_metodo :-
        enviar_a_usuario_intro_tres_metodo,
        declarar_ya_vio_intro_tres,
        solicitar_a_usuario_un_metodo,
        recibir_metodo.

abrir_intro_tres_data :-
        enviar_a_usuario_intro_tres_data,
        declarar_ya_vio_intro_tres,
        solicitar_a_usuario_tipo_data,
        recibir_tipo_data.

abrir_intro_cuatro_pt :-
        enviar_a_usuario_intro_cuatro_pt,
        declarar_ya_vio_intro_cuatro_pt,
        solicitar_a_usuario_un_programa,
        recibir_programa.

abrir_intro_cinco_dg :-
        enviar_a_usuario_intro_cinco_dg,
        declara_ya_vio_intro_cinco_dg,
        solicitar_a_usuario_permiso_para_continuar,
```

```
	recibir_permiso_para_continuar.

iniciar_metodo :-
	soicitar_runtime,
	ejecutar_en_runtime_metodo,
	declarar_process_id_del_metodo,
	obtener_input_stream_del_metodo,
	declarar_input_stream_del_metodo,
	obtener_output_stream_del_metodo,
	declarar_output_stream_del_metodo,
	obtener_error_stream_del_metodo,
	declarar_error_stream_del_metodo,
	declarar_metodo_activo.

procesar_mensaje_input_stream :-
	leer_mensaje_input_stream,
	enviar_usuario_mensaje_metodo,
	solicitar_respuesta_usuario,
	recibir_respuesta,
	declarar_hay_mensaje_del_usuario_para_metodo.

% Pueden existir diversos mensajes para el tutor.
% La estructura general de la definición que los procesa seria:
%	procesar_mensaje_para_el_tutor:-
%		leer_mensaje_para_el_tutor,
%		analizar_contenido,
%		definir_accion.

% Este es el caso en el que el usuario le pide al tutor que reinicie el metodo desde cero.

procesar_mensaje_para_el_tutor:-
	leer_mensaje_para_el_tutor,
	mensaje_es_reiniciar_metodo,
	iniciar_metodo.
% Este es el caso en el que el usuario le pide al tutor que termine la tutoria.

procesar_mensaje_para_el_tutor:-
	leer_mensaje_para_el_tutor,
	mensaje_es_terminar_tutoria,
	enviar_usuario_mensaje_despedida,
	declarar_sesion_inactiva.

procesar_mensaje_error_stream :-
	leer_mensaje_error_stream,
	enviar_usuario_mensaje_error,
	recibir_respuesta,
	declarar_hay_mensaje_del_usuario_para_metodo.

recuperar_metodo :-
	eviar_usuario_mensaje_falla_de_metodo,
	preguntar_usuario_si_desea_iniciar_de_nuevo,
	recibir_respuesta,
	declarar_hay_mensaje_del_usuario_para_tutor.
```

REFERENCIAS BIBLIOGRAFICAS

[AIT-1] A¨lt-Kaci, Hassan. Warren's Abstract Machine A tutorial Reconstruction. MIT Press. 1999

[AMZ-1] Amzi!. Integrating prolog services with C++ Objects.1995

[AMZ-2] Tutorial Amzi! Logic Explorer.Amzi! Inc.1997

[ATK-1] http://www.raleygh.ibm/iag/iahome.html

[BAN-1] Banchilhon. JUDE – Agent. http://jude.sourceforge.net/jude/node5.html

[BAR-1] Baray, Cristobal & Wagner, Kyle. De dónde vienen los agentes inteligentes?. ACM Crossroads Student Magazine

[BAR-2] Baray, Christopher. Evolving cooperation via communication in homogeneous multi-agent systems. 1997.

[BES-1] Besson, J. & Mettler D. Design Principles of Autonomous Agents Seminar “Natürliche und künstliche Intelligenz“. 2000

[BIG-1] Bigus, Joseph et al. Constructing Intelligent Agents using Java. Second Edition. Adisson-Wesley.2001

[BIN-1] BinProlog. http://clement.info.umoncton.ca/BinProlog

[BOO-1] Boone, Barry. Java Essentials for C and C++ programmers. Addison Wesley.1996

[CAL-1] Calejo, Miguel. Java + Prolog. A land of opportunities. PALCP99.

[CAL-2] Calejo, Miguel. InterProlog: a declarative Java-Prolog interface. Declarativa

[CLP-1] F. Mesnard, S. Hoarau, A. Maillard CLP(X) for automatically proving program properties. 1994

[CLP-2] Van Roy, Peter. Issues in implementing Constraint Logic Languagues. DEC Paris Research Lab. 1994

[COD-1] Codgnet, Philippe et al. WAMCC: Complling prolog to C. 1997

[DAV-1] Dávila. Jacinto. Agents in Logic Programming. Thesis of Doctor of Philosophy. 1997

[DAV-2] Davila, Jacinto; Barrios, Alexander and Lopez, Jose. BIOINFORMANTS: BIOlogical, INFORMAtional ageNTS on the Internet. CeSIMO. ULA

[DGK-1] DGKS Prolog. http://www.geocities.com/SiliconValley/Campus/7816/

[FLO-1] Flores-Mendez, Roberto. Hacia una estandarización de los marcos de trabajo para Sistemas Multi-Agentes. ACM Crossroads Student Magazine

[FRA-1] Stan Franklin and Art Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. Institute for Intelligent Systems. University of Memphis

[GOL-1] Goldberg, D. Genetic Algorithms in search, optimization & Machine Learning. Addison-Wesley. 1999

[HER-1] Hermans, Björn. Intelligent Software Agents on the Internet: an inventory of currently offered functionality in the information society & a prediction of (near) future developments. Tilburg University. 1996

[HOG-1] Hogger, Christopher. Essentials of Logic Programming. Clarendon Press. 1990

[INT-1] http://www.declarativa.com

[ISO-1] http://www.logic-programming.org/prolog_std.html

[JAS-1] Jasper. http://www.sics.se/isl/sicstus/sicstus_12.html#SEC162

[JAV-1] javalog. http://www.bird-land.com/java/prolog/index.html

[JAV-2] http://java.sun.com

[JAV-3] http://www.microsoft.com/Java/

[JAV-4] The Java tutorial. http://java.sun.vom/docs/books/tutorial/index.html

[JAV-5] Srinivas, R. Java security evolution and concepts, Part 1. http://www.javaworld.com/javaworld/jw-04-2000/jw-0428-security_p.html

[JAV-6] Java Plug-in HTML Specification. http://java.sun.com/products/plugin/1.3/docs/tags.html

[JIN-1] Jinni. http://www.binnetcorp.com/Jinni/index.html

[JNI-1] http://java.sun.com/j2se/1.3/docs/guide/jni/index.html

[JPR-1] Jprolog http://cs.kuleuven.ac.be/~demoen/jprolog

[KNA-1] Knapik, Michael et al. Developing Intellingent Agents for Distribuited Systems. McGraw Hill Published. 1998

[KOW-1] Kowalski, R & Sadri, F. An Agent Architecture that Unifies Rationality with Reactivity. Imperial College.

[LAN-1] Lance, Danny et al. Programing and deploying java mobile agents with Aglets. Adisson-Wesley.1998

[LEV-1] Levy, M. et al. Translator-Based Multiparadigm Programming.1993

[MIC-1] HOWTO: Make Your Java Code Trusted in Internet Explorer. http://support.microsoft.com/support/kb/articles/Q193/8/77.ASP

[MIN-1] MINERVA. http://www.ifcomputer.com/Products/MINERVA/home_en.html

[MOR-1] Morozov, Alexei. Actor Prolog: an Object-Oriented Language with the Classical Declarative Semantics. Institute of Radio Engineering and Electronics of RAS.

[NAU-1] Naughton, Patrick et al. Java 2: The Complete Reference. Third Edition. McGraw Hill Published. 1999

[NET-1] Object-Signing Tools. http://developer.netscape.com/

[NWA-1] Nwana, H.S. Software Agents: An Overview. Intelligent Systems Research AA&T, BT Laboratories, Ipswich, United Kingdom, 1996. http://www.cs.umbc.edu/agents/papers/ao.ps

[OFI-1] http://www.info.ucl.ac.be/people/PVR/official_report.ps

[OOL-1] http://www.ci.uc.pt/oolpr/oolpr.html

[PAR-1] Parker, Gary. Generating Arachnid Robot Gaits with Cyclic Genetic Algorithms. 1998

[PLO-1] http://www.plogic.com/npp-des.html

[PRO-1] Prolog Café. http://pascal.seg.kobe-u.ac.jp/~banbara/PrologCafe/

[RUS-1] Russell, Stuart et al. Inteligencia Artificial: Un enfoque moderno. Prentice Hall. 1996

[SHO-1] Shoham, Yoav. Agent oriented programming. Computer Science Department. Stanford University. 1990.

[SIC-1] SICStus Prolog http://potato.claes.sci.eg/claes/plugin/npsp.html

[TAR-1] Tarau, P. Low-level Issues in Implementing a High-Performance Continuation Passing Binary Prolog Engine. Dépt. d'Informatique Université de Moncton

[TAR-2] Tarau, P. et al. A Novel Term Compression Scheme and Data.Representation in the BinWAM. Dépt. d'Informatique Université de Moncton

[WOO-1] Wooldridge, M. Issues in Agent-Based Software Engineering. University of Liverpool. 1997.

[WOO-2] Wooldridge, M. On the Source of Complexity in Agent Design. Imperial College

[WOO-3] Wooldridge, M.; Jennings, N. & Kinny, D. A Methodology for Agent-Oriented Analysis and Design. Queen Mary & Westfield College University of Melbourne.

[WOO-4] Wooldridge, M. & Ciancarini, P. Agent-Oriented Software Engineering: The State of the Art. University of Liverpool

[WOO-5] Wooldridge, M. The Logic Rational Agency. Imperial College. 2000

[WOO-6] Wooldridge, M.; Jennings, N. & Kinny, D. The Gaia Methodology for Agent-Oriented Analysis and Design. Queen Mary and Westfield College.

[WPR-1] W-Prolog. http://www.cs.mu.oz.au/~winikoff/wp/

[ZAM-1] Zambonelli, F.; Jennings, N & Wooldridge, M. Organisational Abstractions for the Analysis and Design of Multi-Agent Systems. University of Liverpool