Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Ingeniería Eléctrica

Sección Computación

"Atribución de Significado a Documentos XML con LogCIN-XML"

Tesis que presenta:

Karina Escobar Vázquez

Para obtener el grado de:

Maestra en Ciencias

En la especialidad de:

Ingeniería Eléctrica

Opción:

Computación

Director de tesis:

Dr. José Oscar Olmedo Aguirre

Ciudad de México, D. F. Abril de 2004

Resumen

Desde sus inicios la Web fue enfocada en presentar información comprensible para los seres humanos y se dejó a un lado el hecho de proveer a la información de una semántica formal. El hecho de carecer de una semántica formal dificulta la recuperación precisa de información y la automatización de tareas dentro de la Web. La Web Semántica fue propuesta como la siguiente etapa en la evolución de la Web actual para dotar a la información de significado mediante los lenguajes XML y RDF.

En este trabajo de tesis se propone LogCIN-XML para atribuir de significado a los documentos XML y, de esta manera, poder expresar y ejecutar reglas en lógica de Horn. También se presenta la formalización, diseño e implementación del lenguaje.

Entre las contribuciones principales de este trabajo se pueden destacar:

Las extensiones a XML para incorporar nociones fundamentales de la programación lógica como variables lógicas, procedimientos lógicos y consultas.

La definición de un modelo de representación del conocimiento adecuado para la Web Semántica.

La definición de una arquitectura abierta y distribuida que permite la integración entre aplicaciones, facilitando la interoperabilidad.

Para demostrar algunas de las características más sobresalientes del lenguaje tales como la recuperación de información, la capacidad de deducir información y la generación de nuevos documentos, en este trabajo se aborda y resuelve el problema de la deducción de relaciones de parentesco.

Abstract

From its inception, the Web has been focused in presenting information only understandable for users, leaving out the definition of a formal semantics. The lack of a formal semantics makes difficult to retrieve information that is sparsed on a collection of documents. In this respect, the Semantic Web was proposed as the following stage in the evolution of Web as we know it today. One of the purposes of developing the Semantic Web is in defining a formal meaning to the information by using mark-up languages like XML and RDF.

In this thesis LogCIN-XML is proposed to attribute meaning to XML documents, allowing to express and to execute rules in Horn logic. The formalization, design and implementation of the language are elaborated in some detail in this work.

Among the main contributions of this work we can emphasize:

The XML extentions to incorporate fundamental knowledge of logic programming, logic variables, logic procedures and queries.

The definition of a model for knowledge representation adequate for the Semantic Web.

The definition of an open and distributed architecture allows the integration among applications facilitating their interoperability.

The deduction of kinship relations is presented as a case of study showing some of the most important characteristics of the language among information retrieval, automated reasoning capabilities and XML document generation.

Agradecimientos

Agradezco a mi asesor, el Dr. José Oscar Olmedo Aguirre por su valioso apoyo, dedicación y conocimientos que aportó a este trabajo de tesis. Por la amistad y confianza que me ha brindado.

Agradezco a los miembros del jurado formado por los doctores Guillermo Morales y Sergio Chapa, por el tiempo dedicado a la revisión de la tesis y sus comentarios que han mejorado de una manera substancial este trabajo.

Agradezco a los profesores de la sección que con dedicación, empeño y paciencia aportaron sus conocimientos a mi generación para darle a la sociedad una generación de profesionistas más competentes y comprometidos.

Agradezco a Sofía Reza, el apoyo y la amabilidad que siempre mostró cuando se requería realizar algún trámite administrativo. Además, de la amistad que me brindó y por ver en cada uno de los estudiantes un hijo más.

Al CINVESTAV, ésta gran institución de calidad mundial que me brindó todo el apoyo durante mi estancia.

A CONACYT, por el apoyo económico que brinda a instituciones educativas y por hacer una realidad los deseos de superación de muchos estudiantes mexicanos.

Reconocimientos

A mis padres *Jesús* y *Victoria* por ese gran ejemplo que me han dado, por enseñarme que las cosas grandes requieren grandes esfuerzos, por apoyar siempre mis decisiones, por sus valiosos consejos, por enseñarme los valores de la familia y la responsabilidad, por creer en mí, lo que me ha permitido crecer como ser humano impulsándome a seguir adelante.

A mis hermanos *Yareli*, *Jesús* y *Carlos* por motivarme a dar lo mejor de mí, por escucharme cuando me sentía frustrada y por hacerme reír cuando más lo necesitaba. En especial a *Yareli* por todas las noches que no la dejé dormir platicándole mis proyectos.

A mis grandes amigos *Laura*, *Guadalupe*, *Nubia*, *Verónica*, *Olga*, *Mari*, *Axel* y *Max* por que siempre me apoyaron de una u otra manera en el transcurso de la maestría y por todas esas reuniones que pospusieron para que yo estuviera presente.

A Dios por permitirme ver culminado un deseo.

Índice General

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1 La Web Semántica

La Web fue concebida para diseminar información comprensible para usuarios de todo el mundo. En el año 2002, la Web contenía aproximadamente 3 mil millones de documentos estáticos los cuales eran accedidos por cerca de 300 millones de usuarios. Esta enorme cantidad de datos ha contribuido a acrecentar la dificultad de encontrar, acceder, presentar y mantener la información para un gran número y variedad de usuarios. La principal causa que se puede atribuir como origen de este problema es que la información está escrita en lenguaje natural, solamente comprensible para seres humanos. Este problema fundamental tiene consecuencias inmediatas en la dificultad tanto para mejorar la búsqueda y recuperación de información como para automatizar tareas, como se explica a continuación.

Búsqueda y recuperación de la información: Actualmente, la búsqueda de información se realiza mediante motores de búsqueda como Google, Lycos, Altavista y WebCrawler, entre otros. Los documentos recuperados por este tipo de buscadores se almacenan localmente organizándolos mediante bases de datos. Sin embargo, debido a la pérdida de información de contexto que es inherente a los métodos de búsqueda como TFIDF basados en la frecuencia relativa de términos discriminantes, resulta cada vez menos práctico realizar búsquedas sobre el contenido de documentos utilizando únicamente métodos clásicos de recuperación de la información. La búsqueda es imprecisa porque da como resultado la

recuperación de cientos o miles de documentos que requieren aún de una selección posterior por parte del usuario. Otro problema relacionado involucra el mantenimiento de recursos Web. La responsabilidad de los usuarios para mantener la consistencia es generalmente excesiva. Esta situación ha dado lugar a sitios que contienen información inconsistente o contradictoria. Por otra parte, este tipo de buscadores no proporciona los medios para formular condiciones relativamente complejas sobre el contenido de una colección de documentos ya que no permite definir relaciones entre varios documentos que ayuden a describir, por ejemplo, condiciones de integridad.

Automatización de tareas: La automatización de procesos comerciales que usen la Web como medio de interacción ha sido por mucho tiempo anhelada pero también obstaculizada por los métodos actuales de representación de la información. Los agentes de compra usan heurísticas para extraer y atribuir significado a la información textual escasa o parcialmente estructurada que forma el contenido de los documentos. Sin embargo, los costos de desarrollo y mantenimiento son altos y los servicios proporcionados son limitados. Aunque se han desarrollado una diversidad de tecnologías que van desde CGI hasta JSP o ASP que hacen posible la integración de aplicaciones y la colaboración entre grupos de trabajo, los problemas relacionados con la representación de la información han permanecido sin resolver.

Tim Berners-Lee el creador principal de la Web ya contemplaba estos problemas desde sus inicios, pero debido a las dificultades tecnológicas a las que se enfrentaba en esos días se vio obligado a concebir a la Web basándose en principios sencillos y pragmáticos. En mayo de 2001, Tim Berners-Lee, James Hendler y Ora Lassila publicaron un artículo en la revista Scientific American titulado "The Semantic Web" [26]. En este artículo se describe a la Web Semántica como la siguiente etapa en la evolución de la Web actual. En ella las computadoras no solamente serán capaces de mostrar la información contenida en los documentos como se hace en la actualidad, sino que además serán capaces de atribuir

significado a la información. Para desarrollar la Web Semántica, sus creadores han propuesto el uso de nuevos lenguajes de marcado para introducir estructura y significado al contenido de los documentos. Este tipo de lenguajes permite incluir meta-información en los documentos lo cual conduce a la definición de modelos de datos y semánticos apropiados.

La representación explícita de la semántica de datos, documentos, programas y otros recursos subyacentes hará posible una Web basada en conocimiento que sea capaz de proporcionar nuevos niveles de servicios. Los servicios automatizados mejorarán su capacidad para ayudar a los usuarios a lograr sus metas "entendiendo" mejor el contenido de la Web, proporcionando así categorización y búsqueda de información mediante métodos de razonamiento automatizado. Para que las aplicaciones desarrollen dichas capacidades de razonamiento se requiere la incorporación de lenguajes y de modelos formales de programación, particularmente de aquellos basados en lógica simbólica. El lenguaje de la lógica simbólica permite describir la estructura de la información y atribuirle significado de acuerdo a la aplicación.

1.2 Estructura y Significado de Documentos

Un requisito primordial para interpretar y razonar sobre el contenido de un documento es introducir estructura en los datos. La técnica principal para estructurar información en la Web ha sido la de agrupar el contenido de un documento en elementos organizados jerárquicamente. Esto significa que ciertas secuencias de caracteres contienen información que indica la naturaleza de los fragmentos que forman el contenido del documento. Esta técnica, conocida como *marcado*, describe tanto el diseño de los datos del documento como de su estructura lógica, permitiendo organizar y localizar más eficientemente la información.

Un lenguaje de marcado especifica: (i) qué elementos pueden usarse, (ii) cómo pueden combinarse y, (iii) cómo pueden interpretarse por programas y aplicaciones.

HTML es el lenguaje mejor conocido y más ampliamente usado en la Web. Provee elementos para especificar la presentación abstracta de la información. Los navegadores (*browsers*) usan ésta información para formatear apropiadamente el contenido de los documentos Web. Así, la única información que provee HTML es sobre la presentación del documento. Casi siempre, esta información no es suficiente para automatizar los servicios avanzados que se prevén en la Web Semántica. Las aplicaciones necesitarán de marcados especializados que especifiquen el papel que las partes del contenido tienen con relación a ellas. Para esto es necesario especificar el significado de los elementos.

XML (*eXtensible Markup Language*) es un metalenguaje que permite diseñar una jerarquía de lenguajes de marcado mediante la definición de colecciones de elementos. Los elementos de XML permiten organizar el contenido de un documento mediante meta-información que describe sus características. Por ejemplo, los datos contenidos en un archivo de configuración de un programa se pueden organizar y describir agrupándolos mediante elementos XML. La popularidad que XML ha adquirido en los últimos años se debe a la facilidad con la que programas y aplicaciones intercambian datos, la cual ha dado lugar a su adopción como un estándar para la representación, el almacenamiento y la transferencia de información.

Sin embargo, para que un programa pueda procesar un documento XML, el documento debe estar bien construido. Se dice que un documento XML está *bien construido* cuando la composición de sus elementos forma una estructura de árbol multicamino, ordenado y etiquetado. Por otra parte, un documento XML es *válido* cuando la composición sigue reglas bien definidas dadas por un DTD. Un DTD (*Data Type Definition*, por sus siglas en inglés) es una declaración de tipo de documento mediante el cual se define la gramática de una colección de documentos. La ventaja de esta interpretación abstracta de un documento XML como árbol es que hace posible codificar todos los tipos de estructuras de datos en una sintaxis libre de ambigüedades; su desventaja es que no especifica el uso que los datos

tendrán para la aplicación ni ninguna otra forma de interpretación. De esta forma, las aplicaciones que usan XML para el intercambio de datos deben acordar de antemano el vocabulario usado y su significado. Por otra parte, RDF (*Resource Description Framework*, por sus siglas en inglés) es un lenguaje para representar información de recursos en la Web así como sus relaciones. RDF se basa en la idea de identificar recursos mediante URIs (Universal Resource Identifiers, por sus siglas en inglés) y describir recursos mediante propiedades simples junto con sus valores correspondientes.

Aunque el problema de la atribución de significado al lenguaje que usan las aplicaciones se puede resolver parcialmente utilizando juntos XML y RDF [2], la unión de ambos lenguajes carece del suficiente poder expresivo que permita especificar los axiomas, condiciones y restricciones que describan las características fundamentales de un sistema.

Por las ventajas que ofrece tanto para la recuperación de la información como para la automatización de tareas, la Web Semántica irá remplazando paulatinamente a la Web como aún la conocemos hoy en día. Esta tendencia aumentará la necesidad por disponer de lenguajes más expresivos y modelos formales más poderosos que sean el fundamento de sistemas distribuidos de información basados en la Web.

1.3 Planteamiento del problema

El problema que se plantea en esta tesis consiste en definir métodos de búsqueda y recuperación de información incorporados en un lenguaje que permita la representación del conocimiento y que mejore considerablemente la expresividad y la eficacia de consultas que involucran colecciones de documentos.

Tim Berners-Lee ha señalado cuáles son las características que debe poseer un lenguaje apropiado para la Web Semántica[2]:

Sintaxis reducida.

Semántica bien definida.

Suficiente poder expresivo para representar conocimiento.

Mecanismos de razonamiento eficaces pero comprensibles.

Métodos para construir bases de conocimientos grandes.

El lenguaje debe facilitar la representación declarativa del conocimiento mediante un proceso que involucra la identificación, definición, formalización y aceptación consensuada de conceptos primordiales organizados en ontologías. La definición del lenguaje debe estar acompañada de un modelo semántico que permita formalizar y automatizar el proceso de recuperación de la información mediante un proceso eficiente de inferencia. Finalmente, el sistema debe mostrar un diseño abierto que asegure la interoperabilidad con otras aplicaciones distribuidas aprovechando la infraestructura disponible de la arquitectura cliente-servidor de Internet.

1.4 Propuesta de solución

Los sistemas basados en reglas se han usado tradicionalmente en diversas áreas de la computación que incluyen los compiladores, las bases de datos, la programación lógica y la inteligencia artificial. Sin embargo, en la Web Semántica, las reglas se han estudiado menos que las ontologías (o más bien taxonomías). En este trabajo se intenta remediar esta situación explorando sistemas de reglas adecuadas para la Web Semántica, su sintaxis, semántica, eficiencia, transformación y compilación. El tipo de reglas que se consideran son las cláusulas de Horn (procedimientos lógicos) como en la programación lógica. Las reglas pueden usarse para mejorar el contenido de páginas Web o de documentos XML en varias formas, por ejemplo, permitirán la inclusión dinámica de conclusiones derivadas del contenido de colecciones de documentos.

El desarrollo de ontologías para la Web daría lugar al desarrollo de un lenguaje uniforme ontológico con una lógica descriptiva de la taxonomía y de reglas

en lógica de Horn. Las taxonomías a gran escala como aquellas en DAML+OIL requerirán un sistema de reglas para derivar o usar cierta información implícita que no puede capturarse solamente por la taxonomía. Las reglas requeridas pueden describirse de acuerdo a una lenguaje suficientemente expresivo. Las taxonomías DAML+OIL y las reglas deberán expresarse en formas compatibles en un lenguaje abierto.

Las combinaciones posibles entre taxonomías y reglas deben compararse sistemáticamente con respecto a los criterios de naturalidad contra formalidad, expresividad contra eficiencia y compatibilidad de tipos de las variables en las reglas contra tipos de los términos en las premisas. Sin embargo, las mayores dificultades pueden surgir por las combinaciones entre la forma de la taxonomía y el lenguaje de reglas. Por ejemplo, las lógicas con sistemas de tipos (*many-sorted logics*) combinan bien con las lógicas de Horn y aún con otras lógicas de primer orden como se demuestran en las soluciones del problema del apisonador de Schubert (Schubert's steamroller) cuyo éxito radica en reducir el espacio de búsqueda. Por lo tanto, se espera que el sistema de tipos (*sorts*) esté organizado para que se identifique naturalmente con la taxonomía. Si una comunidad de usuarios puede establecer sus requerimientos con respecto a la expresividad de la taxonomía, de las reglas o de ambos, entonces será más fácil identificar la combinación apropiada.

Puesto que no se puede anticipar los requerimientos de comunidades de usuarios y desarrolladores, para resolver el problema que se plantea en esta tesis, parece más sensato separar completamente el diseño del lenguaje de reglas de cualquier taxonomía. Esta separación de responsabilidades permitirá que el lenguaje esté disponible para el mayor número de aplicaciones pero anticipando que será usado eventualmente para representar conocimiento ontológico.

El lenguaje propuesto, LogCIN-XML, es un primer paso para desarrollar sistemas de conocimiento ontológico en la Web que permitan expresar y ejecutar reglas en lógica de Horn. LogCIN-XML se ha diseñado como un lenguaje basado en

reglas, distribuido, abierto e inter-operable entre aplicaciones y plataformas operativas de la Web. Estos criterios de diseño permitirán compartir o intercambiar datos y reglas entre componentes distribuidos de software en la Web, sistemas heterogéneos cliente-servidor, sistemas de comercio electrónico, etc.

En la Fig. 1 se visualiza una arquitectura general de las aplicaciones que utilizan LogCIN-XML. Esta arquitectura tiene una organización por niveles. En el nivel más bajo se encuentran la colección de documentos Web. En el siguiente nivel se ubica la base de documentos XML cuyo propósito es describir a los documentos Web del nivel inferior. Sobre la capa XML se encuentra la infraestructura de coordinación y comunicaciones que ofrece java y de razonamiento de Prolog Café [16]. En el siguiente nivel se ubica LogCIN-XML, el cual representa a cada documento XML como un término lógico sin variables. Esta representación permite a la máquina LogCIN-XML derivar colusiones sobre el contenido de los documentos. Finalmente, en el nivel superior se encuentran las aplicaciones como sistemas interrogadores, scripts, formateo de documentos, entre otros.

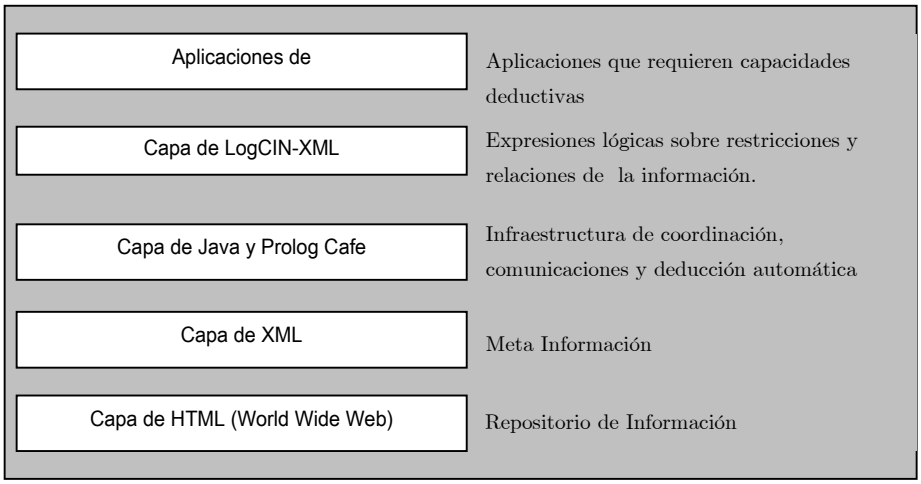

Fig. 1 Organización por capas de una aplicación basada en LogCIN-XML

La naturalidad del lenguaje de reglas de Horn como lenguaje para la representación del conocimiento descansa en las relaciones que se pueden establecer:

Entre la noción de término simbólico en lógica con la de documento XML.

Entre la especificación de las propiedades de un conjunto de términos simbólicos con la especificación de las restricciones de integridad que deben cumplir las colecciones de documentos XML.

La relación de término simbólico y documento XML se hace evidente al observar que un término simbólico es una instancia de la gramática abstracta de un lenguaje y que puede identificarse, por lo tanto, con un árbol de sintaxis multicamino, ordenado y etiquetado. En forma similar, un documento XML posee una estructura bien definida que también puede representarse por un árbol de sintaxis con las mismas características. De esta forma se puede establecer una correspondencia entre ambos conceptos, siendo el árbol de sintaxis la representación abstracta que los relaciona. En la Fig. 2 se muestran las relaciones entre el árbol de sintaxis de un término con el árbol de sintaxis de un documento XML correspondiente.

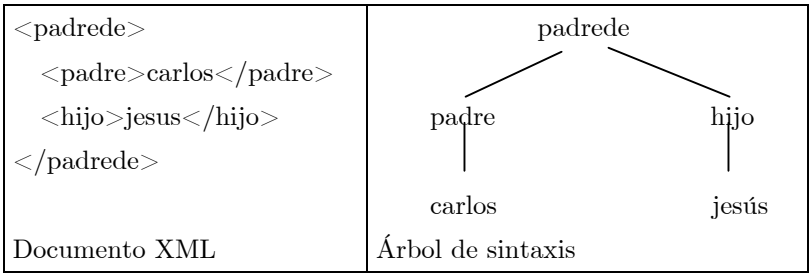

Fig. 2 Ejemplos de un árbol de sintaxis para el término padrede(padre(carlos),hijo(jesus))

En cuanto a la relación entre ambas interpretaciones, la interpretación de validez semántica de una colección de documentos es similar a aquella de la semántica declarativa de la programación lógica. Esta interpretación consiste en

definir primero la colección de todos los documentos XML sintácticamente válidos que se pueden generar por medio de la composición de otros documentos válidos (*universo de Herbrand*). Después se clasifica esta colección de documentos de acuerdo a las propiedades básicas que deben cumplir con respecto a la especificación del sistema (*base de Herbrand*). A continuación se construyen nuevos documentos a partir de aquellos que ya cumplen la especificación usando las cláusulas de Horn como reglas de derivación. Este proceso de generación de nuevos documentos continúa hasta alcanzar la colección completa de todos los documentos que cumplen la especificación. La colección de documentos XML así generada constituye un modelo para la especificación del sistema y es el principio de la semántica declarativa de LogCIN-XML. De acuerdo a ésta interpretación, una consulta se puede contestar siempre que sea consecuencia lógica del contenido de la colección de documentos y de las reglas de Horn de la especificación.

1.5 Objetivo general

El objetivo general de este trabajo de tesis es desarrollar un lenguaje y un modelo formal que permite atribuirle significado a una colección de documentos XML.

1.6 Objetivos específicos

1. Definir formalmente la correspondencia sintáctica entre documento XML y términos lógicos, así como predicados sobre ellos (construyendo el universo de Herbrand y la base de Herbrand). Intuitivamente esto equivale a considerar todos los documentos válidos (sintácticamente) que se pueden construir a partir de fragmentos o piezas de información multimedia.
2. Definir formalmente la semántica usando la construcción de punto fijo sobre la base de Herbrand. Intuitivamente la construcción de punto fijo permite:
    a. Decidir cuáles son los documentos iniciales que cumplen con la especificación (hechos).

   b. Decir cuáles son los documentos que pueden construirse a partir de los documentos iniciales previamente construidos de ésta forma (reglas).
3. Diseñar e implementar el sistema. Integrar métodos de manipulación de contenido de documentos al procedimiento de inferencia usado en Prolog de tal manera, que conserve la interpretación abstracta de un documento.

1.7 Metodología

La relación entre la semántica declarativa y su interpretación operacional que caracteriza a la programación lógica se utiliza como guía metodológica para establecer el plan a seguir en este trabajo. Los pasos a seguir son:

1. Definir la sintaxis del lenguaje en cláusulas de Horn y XML.
2. Establecer la correspondencia entre términos y expresiones MXL.
3. Establecer la correspondencia entre predicados y elementos XML.
4. Establecer la semántica declarativa del lenguaje.
5. Definir la interpretación procedural del lenguaje basándose en aquella de Prolog.
6. Validar resultados.

1.8 Contribuciones

LogCIN-XML está construido sobre una máquina de inferencia por lo que se puede afirmar que LogCIN-XML está dirigido principalmente a todas las aplicaciones que requieran capacidades deductivas.

Mediante LogCIN-XML se puede proveer de significado a la información localizada en los documentos XML, esto se debe a que ahora contamos con procedimientos y cláusulas, y no sólo con datos estructurados. Mediante los procedimientos se puede derivar información lógicamente a partir de las colecciones de documentos, se puede definir restricciones lógicas y provee las bases para que en un futuro se pueda mover información entre documentos de forma automática

utilizando los métodos de satisfacción de restricciones de Prolog y las reglas ECA (evento – condición - acción).

1.9 Organización del documento

Esta tesis está organizada de la siguiente forma. En el capítulo 2 se presentan los antecedentes del sistema incluyendo una breve descripción de las tecnologías usadas como HTML, XML, lenguajes ontológicos, el lenguaje de programación lógica Prolog así como las varias implementaciones de Prolog en Java. En el capítulo 3 se describe el caso de estudio, el problema de la representación de relaciones de parentesco el cual es usado a lo largo de la tesis. En el capítulo 4 se presenta la formalización del lenguaje LogCIN-XML como un lenguaje lógico de primer orden comparable al lenguaje de cláusulas de Horn. En el capítulo 5 se presenta el diseño del sistema desde el punto de vista de su arquitectura de software. En el capítulo 6 se explica la implementación del sistema en la arquitectura cliente-servidor de Internet. Finalmente, en el capítulo 7 se presentan las conclusiones del trabajo y se discuten brevemente el trabajo futuro.

Capítulo 2

Antecedentes

En este capítulo se muestra una breve descripción de las tecnologías de la Web y cómo surgió la necesidad de proponer un lenguaje de marcado diferente para construir la Web de la siguiente generación (Web Semántica). El lenguaje propuesto para la construcción de la Web es XML pero presenta ciertas carencias que ha ocasionado el desarrollo de otros lenguajes; también se revisan brevemente algunos de los más importantes lenguajes que han surgido. Además, se puntualizan brevemente algunas implementaciones de Prolog que son de suma importancia para el desarrollo de esta tesis.

2.1 Lenguaje de Marcado de Hipertexto HTML

En 1989, Tim Berners-Lee propuso compartir la información en el marco del CERN, el Centro Europeo de Investigación Nuclear, utilizando documentos de texto de hiperenlace. Un colega llamado Anders Berglund, uno de los primeros en adoptar SGML (Standard General Language), le aconsejó que utilizara una sintaxis de tipo SGML. A partir de un simple ejemplo de tipo de documento según la norma SGML desarrollaron una versión de hipertexto llamada Lenguaje de marcado de hipertexto o HTML, por sus siglas en inglés.

Comparado con los 20 años de trabajo que necesitó SGML, HTML se definió rápidamente cumpliendo con sus expectativas. Berners-Lee denominó a su sistema de hipertexto *World Wide Web*, la red de alcance mundial el cual hoy día es el sistema de información de hipertexto más popular y diversificado. Muchos piensan que su sencillez es la clave de su éxito. En cualquier caso, la sencillez de HTML y

de otras especificaciones Web explica la rápida proliferación de sistemas y herramientas diseñadas para la Web.

HTML ha heredado algunas de las mayores virtudes de SGML. Salvo escasas excepciones, sus tipos de elementos son generalizados y descriptivos, y no elementos de formato similares a los de lenguajes como Tex y Microsoft Word. Esto implica que los documentos HTML se pueden mostrar en pantallas de texto, bajo interfaces gráficas de usuario, e incluso ser proyectados a través de altavoces imperceptibles para la vista.

Los documentos HTML utilizan para el marcado la convención simple de paréntesis de SGML. De esta forma, los autores pueden crear documentos HTML en casi cualquier editor o procesador de texto. Los documentos son además compatibles con casi todos los sistemas de computadoras existentes.

Por otro lado, HTML sólo utiliza una serie fija de tipos de elementos, ningún tipo de documento sirve para todos los propósitos, por tanto, HTML sólo adoptó la primera afirmación de SGML, esto es, que las representaciones de documentos deben normalizarse. HTML no es extensible, por consiguiente no puede adecuarse a ciertos tipos de documentos, y no ha acabado de definirse del todo. Cuando HTML dispuso de un DTD formal, ya existían miles de páginas Web con HTML erróneo.

A medida que la popularidad de la Web fue en aumento, el tipo de documento fijo de HTML empezó a mostrar numerosos inconvenientes. Los vendedores de navegadores vieron una oportunidad de aumentar su cuota de mercado comercializando extensiones incompatibles con HTML. La mayoría de ellas formateaban las órdenes, por tanto, suponían una merma de la interoperabilidad de la Web. Por ejemplo, CENTER, elemento de Netscape no puede ser “pronunciado” en un texto por el sintetizador de voz. Algunas computadoras no pueden recrear un elemento BLINK. Hasta cierto punto este problema fue consecuencia de las limitaciones de HTML.

Puesto que el marcado de formato registrado representaba un creciente peligro para la interoperabilidad y escalabilidad de la Web, el Consorcio de la Web Mundial, presidido por Berners-Lee decidió tomar medidas. Atacaron el problema por tres flancos.

En primer lugar, se acordó adoptar la convención SGML para adjuntar formateo a documentos, es decir, la hoja de estilos. Inventaron un sencillo lenguaje de hoja de estilos específico de HTML llamado CSS (Cascading Style Sheets), que permitía adjuntar el formato de documentos HTML sin por ello asumir su marcado característico registrado y orientado a la presentación.

En segundo lugar, se ideó un mecanismo simple para añadir abstracciones a HTML. Permitía inventar nuevas abstracciones pero no integraba mecanismo alguno que las hiciera posible. Sin embargo, nadie dudaba que ésta situación acabaría por ser insostenible.

Y finalmente, se acordó que los tipos de documentos se definieran formalmente para que, de esta forma, se pudiera cotejar su validez.

2.2 Lenguaje Extensible de Marcado XML

El Consorcio de la Web al ver los problemas que representaba continuar con HTML decidió desarrollar una subconjunto de SGML que aunara las principales ventajas de SGML con la sencillez de la Web. El nuevo lenguaje recibió el sugerente nombre de lenguaje extensible de marcado XML (eXtensible Markup Language). Decidieron crear asimismo normas asociadas para definir hiperenlaces avanzados y hojas de estilos.

XML es una especificación. Un documento XML es un medio estructurado para almacenar información. El bloque de construcción básico de un documento XML es la entidad, que contiene datos analizados o no sintácticamente.

Los componentes de marcado de XML son:

- Etiquetas de elementos.

- Instrucciones de procesamiento.
- Declaraciones de tipos de documento.
- Referencias de entidades.
- Comentarios.
- Secciones marcadas.

Etiquetas

Las etiquetas se emplean para describir elementos. Por ejemplo, el elemento ciudad está formado por las etiquetas <ciudad></ciudad>. Los elementos XML pueden estar vacíos, lo cual significa que no contiene datos de caracteres analizados sintácticamente.

Los elementos son los responsables de establecer la estructura lógica de un elemento como unidad lógica de información, mientras que un atributo es una característica de esa información. En otras palabras, un elemento representa un objeto de información mientras que un atributo representa una propiedad de ese objeto.

Instrucciones de procesamiento

Las instrucciones de procesamiento siempre comienzan con un signo menor que seguido de un signo de interrogación (<?) y terminan con un signo de interrogación seguido de un signo de mayor que (?>). Por ejemplo, <? Xml versión=”1.0”?> es una instrucción de procesamiento muy usada. Esta instrucción de procesamiento indica que el documento se basa en la versión 1.0 de XML. Los analizadores XML pasan las instrucciones de procesamiento a la aplicación.

Declaraciones de tipo de documento

Las declaraciones de tipo de documento se emplean en XML para especificar información acerca de un documento, incluyendo el elemento raíz del mismo y la definición de tipo de documento (DTD). La declaración de tipo de documento es muy importante a la hora de establecer si un documento es válido o si sólo está

bien construido. Las tareas que lleva a cabo la declaración de tipo de documento son:

- Especificar el elemento raíz del documento.
- Definir los elementos, atributos y entidades específicas del documento.

Un documento bien construido es aquel que se adhiere a las reglas de la sintaxis de XML, mientras que un documento válido es un documento bien construido que se adhiere a una DTD.

Entidades y referencias de entidades

Las entidades representan una unidad de almacenamiento y todo documento XML tiene por lo menos una entidad conocida como entidad de documento. Existen varios tipos de entidades, a continuación hablamos de algunos tipos más comunes.

Las entidades generales están diseñadas para ser utilizadas en el contenido del documento y nos permite aislar una cadena de texto reutilizable. Una entidad general se define por <!ENTITY nombre_entidad definición_entidad> y se hace referencia a ellas por medio del nombre de la entidad entre un ampersan y un punto y coma.

Una entidad de parámetro es una entidad general que sólo se una en una DTD. Este tipo de entidades se utilizan para ayuda a modularizar la estructura de una DTD almacenando componentes de declaración habitualmente utilizados. Una entidad de parámetro se define por <!ENTITY % nombre_entidad definición_entidad> y se hace referencia a ellas por medio del nombre de la entidad entre un signo porcentual y un punto y coma.

La mayoría de referencias de entidades tienen que ser declaradas de forma explícita antes de poder ser utilizadas.

Comentarios

Los comentarios comienzan con <!-- y terminan con -->. No se puede incluir guión doble (--) dentro de un comentario. Los comentarios pueden aparecer en una sección CDATA, en cuyo caso no se tratan como comentarios.

Secciones CDATA

Las secciones CDATA (Secciones de datos de caracteres) se emplean en los documentos XML para poner texto que no se debe identificar como datos de caracteres XML. Por ejemplo, <![CDATA[<nombre>Carlos</nombre>]]> indica que el elemento <nombre> no se reconoce como marcado XML.

2.2.1 Esquemas y DTDs

Los esquemas se usan en XML para modelar una clase de datos. Una vez que un modelo de datos está colocado para una determinada clase de datos, es posible crear documentos XML estructurados que se adhieren a ese modelo.

Un esquema describe la organización de los datos de marcado y de los caracteres en un documento XML válido. Además, describe un vocabulario XML específico definiendo claramente las relaciones entre los elementos del mismo.

Los esquemas establecen restricciones en la estructura del contenido del documento de dos maneras:

- Establece el modelo de contenido de los documentos, lo que define el orden y la anidación de los elementos.
- Establece los tipos de datos usados en el documento.

Los DTD son una forma de establecer un esquema para los documentos XML. Pero los DTD usan una sintaxis especializada que describe la estructura de los vocabularios XML. La sintaxis DTD es compacta, pero la forma en que describe la estructura de los documentos es bastante crítica y poco intuitiva.

XML Schema emplea un vocabulario XML llamado XML-Data. Un esquema XML es algo muy parecido, en lo que a finalidad se refiere a una DTD. Los

esquemas XML describen elementos y sus modelos de contenido, de forma que los documentos puedan ser válidos. Además, permiten asociar los tipos de datos con elementos.

2.2.2 Documentos válidos y bien construidos

Para que un documento se considere bien construido deberá:

- Tener un elemento de documento (elemento raíz) en el que se hallen todos los demás elementos.
- Poseer una etiqueta vacía en todos los elementos que no poseen subestructura.
- Contar con un par de etiquetas de inicio y cierre en todos los elementos que poseen subestructura.
- Tener todos los elementos correctamente anidados.
- Tener todos los valores de atributo entre comillas.
- Declarar todas las entidades que utilice en los DTD o XML Schema.

Un documento es válido cuando cumple:

- El documento debe estar bien construido.
- El nombre del elemento raíz del documento debe coincidir con el nombre de la declaración de tipo de documento.
- El documento debe tener una DTD o XML Schema que declare todos los elementos, atributos y entidades que se utilicen en el documento.
- El documento debe adherirse a la gramática que establezca la DTD o XML Schema.

2.2.3 Hojas de estilo

Las hojas de estilo permiten adjuntar un estilo visual propio a los documentos sin modificar el marcado generalizado. Como el estilo se describe en una identidad aparte, llamado hoja de estilo, el software que no tenga interés en el estilo sólo se desentiende de éste. Las hojas de estilo se codifican por medio de CSS o de XSL (eXtensible Style Language). Una hoja de estilos define reglas de diseño que le

indican a un navegador cómo mostrar partes de un documento. Dado que XML es un metalenguaje basado en contenido depende completamente de las hojas de estilos para su visualización.

2.3 Trabajos Relacionados con la Web Semántica

Para construir la Web Semántica, Tim Berners-Lee propuso XML [3]. XML es un lenguaje de marcado que permite añadir estructura a los documentos, pero no dice nada acerca del significado de la información incluida en el documento. El problema que se tiene con XML es que permite a la misma unidad semántica ser expresada por más de una estructura sintáctica. Esto se resuelve utilizando la combinación de XML y RDF. Pero esta combinación de lenguajes carece de poder expresivo y dificulta la tarea de modelar conocimiento. Se han realizado varios trabajos orientados a la Web Semántica utilizando lenguajes ontológicos debido a que estos lenguajes están orientados a modelar conocimiento. En esta sección se revisan algunos de los más importantes lenguajes ontológicos. Además, se realiza una revisión de los lenguajes de consulta que han surgido como consecuencia de la necesidad de tener herramientas que permitan consultar datos de grandes colecciones de documentos XML.

2.3.1 Lenguajes Ontológicos

Una ontología es una especificación explícita y legible por máquinas de una conceptualización alcanzada por consenso [1]. Las ontologías se usan en sistemas de agentes, sistemas de manejo de conocimiento y plataformas de comercio electrónico. Además, también pueden generar lenguaje natural, integrar información de forma inteligente y extraer información de textos, pero las ontologías no sólo son importantes en aplicaciones donde el conocimiento tiene un rol importante, también pueden aplicarse en la Web Semántica; ya que ha sido definida como “la estructuración conceptual de la Web de una forma explicita y legible por máquinas”.

Se han desarrollado lenguajes ontológicos que extienden XML y RDF como se describe en [1]. Entre estos lenguajes encontramos a: XOL[1], SHOE[1], OML[6], RuleML[5] y XDD[2] los cuales se basan en XML. OIL[6] y DAML+OIL[1] se basan en XML y RDF. Algunos de estos lenguajes tienen correspondencia con teorías formales de la lógica de primer orden.

RDF (Resource Descriptive Framework)

RDF fue creado por el W3C para describir los recursos de la Web. Este lenguaje permite especificar la semántica de los datos basados en XML de una forma estandarizada e inter-operable. También, provee mecanismos para representar explícitamente servicios, procesos y modelos de negocios, además permite reconocer información no explícita.

El modelo de datos de RDF es equivalente al formalismo de las redes semánticas. Éste consiste de tres tipos de objetos:

- *Recursos.-* Los recursos se describen mediante expresiones RDF y siempre se nombran por un URI (Unifided Resource Identified) más un identificador opcional, consecuentemente RDF puede describir cualquier cosa que se pueda identificar mediante un esquema URI y no sólo objetos en la Web (tales como páginas, partes de páginas, o colecciones de páginas).
- *Propiedades.-* Las propiedades se usan para describir un recurso ya que definen aspectos específicos, características, atributos o relaciones del recurso.
- *Declaraciones.-* Las declaraciones permiten asignar valores a las propiedades de un recurso específico (este valor puede ser una declaración RDF) y son ternas objeto-atributo-valor.

El significado en RDF proviene de la especificación de términos y conceptos que se definen por los URIs y los nombres. Debido a que los URIs son únicos, dos

sistemas pueden definir algunos conceptos y cada uno usar un URI diferente para referirse al sistema y de esta manera evitar colapsos. Sin embargo, dos sistemas que acuerdan compartir conceptos usarán el mismo URI para que efectivamente compartan la semántica.

El modelo de datos de RDF también define algunos constructores de meta nivel (tales como contenedores de tipos para describir colecciones de recursos) y declaraciones de orden superior (declaraciones acerca de las declaraciones). Las declaraciones de orden superior se modelan en RDF y permiten la representación de modalidades tales como creencias. El modelo de datos RDF no provee un mecanismo para la definición de relaciones entre las propiedades de los recursos (atributos) y recursos ya que esta tarea corresponde a RDFS (*Resource Descriptive Framework Schema*). RDFS ofrece primitivas para definir modelos de conocimiento. RDFS se usa como un formato de representación en varias herramientas y proyectos, como son Amaya, Mozilla y SilRI, entre otros.

RDF utiliza XML para las expresiones sintácticas de las instancias del modelo, que es un recurso. RDF es esencialmente un modelo de datos y no remplaza XML. En su lugar, construye una capa sobre XML, haciendo posible el intercambio inter-operable de información semántica.

SHOE (Simple HTML Ontology Extension)

SHOE fue desarrollado en la Universidad de Maryland, fue creado como extensión de HTML incorporando conocimiento semántico en los documentos HTML y en otros documentos de la Web. Actualmente, SHOE está siendo adaptado a la sintaxis de XML. SHOE hace posible que los agentes recolecten información significativa de las páginas Web y documentos, mejorando los mecanismos de búsqueda y recolección de conocimiento. Este proceso consiste de tres fases: (i) definir una ontología, (ii) comentar páginas HTML con información ontológica que las describa y (iii) tener un agente que recupera la información

semánticamente por la búsqueda en todas las páginas existentes manteniendo la información actualizada.

OML (Ontology Markup Language)

OML fue desarrollado en la Universidad de Washington y está parcialmente basado en SHOE, por esto OML y SHOE comparten varias características y se encuentra formado por cuatro capas. La capa OML Core está relacionada a los aspectos lógicos del lenguaje y está incluida en el resto de las capas. Las capas restantes son Simple OML la cual realiza la correspondencia directamente a RDF; Abreviated OML incluye conceptos de teoría de gráficas y Standard OML que es la versión más expresiva de OML.

XOL (XML-Based Ontology Exchange Language)

XOL fue diseñado por la comunidad americana de bio-informáticos para el intercambio de definiciones ontológicas entre un conjunto de sistemas heterogéneos. Sus desarrolladores primero estudiaron las necesidades de representación de información de los bio-informáticos y seleccionaron a los lenguajes Ontolingua y OML como base para su creación.

OIL (Ontology Interchange Language)

OIL fue desarrollado en el proyecto OntoKnowledge y permite la interoperabilidad semántica entre los recursos de la Web. Su sintaxis y semántica se fundamentan en los lenguajes basados en formas, provee un modelo de primitivas comúnmente usadas en los planteamientos de la ingeniería ontológica y la semántica formal, el razonamiento lo encuentra en los métodos de la descripción lógica. OIL está construido sobre RDF y tiene las siguientes capas: Core OIL contiene grupos de primitivas OIL que realizan el mapeo directo a RDF; Standard OIL utiliza más primitivas que las definidas en RFD y es el modelo OIL completo;

Instante OIL agrega instancias de conceptos y funciones al modelo previo y Heavy OIL es la capa para las futuras extensiones de OIL.

Las principales primitivas de modelado son clases (conocidos como formas) con propiedades (conocidos como ranuras). Una forma provee un contexto para modelar una clase que es generalmente definida por la relación subclase-de, las clases usan las ranuras para especificar restricciones adicionales en instancias de la nueva clase.

DAML+OIL (DARPA Agent Markup Language + OIL)

DAML+OIL está siendo desarrollado por un comité de Estado Unidos y la Unión Europea para permitir la interoperabilidad en XML. La especificación inicial fue llamada DAML-ONT. Actualmente, lleva el nombre de DAML+OIL debido a que se basó en OIL, porque comparten los mismos objetivos. DAML+OIL extiende RDF y ha capturado las primitivas de modelado comúnmente usadas en el modelado orientado a objetos. Además, integra el sistema de formas y algunos conceptos de esquemas.

XDD (XML Declarative Description)

XDD es un lenguaje que hace posible la representación semántica de los recursos de la Web, emplea la sintaxis de XML y mejora las expresiones al utilizar la teoría de la descripción declarativa. Una descripción en XDD es un conjunto de elementos ordinarios XML que extiende los elementos XML con variables y relaciona los elementos XML en términos de cláusulas. Un elemento ordinario XML denota una unidad semántica y es un sustituto de una pieza de información en el dominio de una aplicación real. Un elemento extendido XML representa información implícita o un conjunto de unidades semánticas. Las cláusulas representan reglas, relaciones condicionales, restricciones de integridad, y axiomas ontológicos. Para traducir descripciones XDD a documentos XML convencionales,

los desarrolladores de XDD sugieren utilizar los editores y analizadores sintácticos de XML.

RuleML (Markup Language Rules)

A pesar de que las reglas se han usado extensamente en la teoría de la ciencia de la computación, la tecnología de compiladores, las base de datos, la programación lógica y la inteligencia artificial, pero dentro de la Web Semántica han estudiado y usado menos en comparación con las ontologías. Las reglas pueden usarse para mejorar el contenido de las páginas de la Web y de los documentos XML en varias formas. Por ejemplo: las reglas de derivación permiten la inclusión dinámica de hechos derivados, mientras las reglas de reacción permiten la especificación del comportamiento en respuesta a los eventos del navegador. La iniciativa de RuleML es el desarrollar un lenguaje de reglas basado en XML/RDF y abierto. RuleML [5] ofrece una sintaxis XML para reglas de representación de conocimiento, interoperabilidad entre la mayoría de sistemas de reglas comerciales y no comerciales. RuleML tiene un diseño modular y combina desde reglas de reacción (reglas evento-condición-acción), reglas de integridad (reglas consistencia-vigilancia), reglas de derivación (reglas implicación-inferencia) y hechos.

La siguiente tabla es una versión condensada de aquellas presentadas en [1] y se resume el estudio comparativo de los lenguajes propuestos para la Web Semántica. El signo + indica que la característica es soportada por el lenguaje, mientras que el signo – indica que no lo es. Cuando la característica requiere mayor explicación se indica con +-.

Conceptos	XOL	SHOE	OML	RDF(s)	OIL	DAML+OIL	LogCIN-XML
Generalidad							
Particiones(clase)	-	-	+	-	+	+	+-
Atributos							
De instancia	+	+	+	+	+	+	-
De clase	+	-	+	+	+	+	-
Ámbito local	+	+	+	+	+	+	-
Ámbito global	+	-	+	+	+	+	-
Restricciones							
Preestablecidas	+	-	-	-	-	-	+-
De tipo	+	+	+	+	+	+	+-
De cardinalidad	+	-	-	-	+	+	+-
Taxonomías							
Subclase de	+	+	+	+	+	+	+-
Descomposición exhaustiva	-	-	+-	-	+	+	+-
Descomposición disjunta	-	-	+	-	+	+	+-
Relaciones/Funciones							
Relaciones/Funciones	+-	+	+	+-	+-	+-	+
Restricciones de tipo	+	+	+	+	+	+	-
Restricciones de integridad	-	-	+	-	-	-	+
Axiomas							
Lógica de Primer orden	-	+-	+	-	+-	+-	+
Lógica de Segundo orden	-	-	-	-	-	-	+-

Tabla 1. Comparación de características.

En la tabla 1 en el concepto generalidad se observa que la noción de partición (o agrupamiento) de objetos en clases disjuntas es una característica que sólo ofrecen OML, OIL y DAML+OIL.

En el concepto de atributos se observa que la mayoría de los lenguajes ontológicos distinguen entre atributos de instancia y de clases, también especifican ámbitos locales y globales para los nombres de clases [1]. XOL usa marcas propias y plantillas para expresar atributos de clases e instancias. Por el contrario, SHOE y RDF no distinguen entre instancias y atributos de clase; tampoco es posible

establecer un valor preestablecido para los atributos de clase. Tanto en OML como en OIL no se distingue entre la sintaxis de las clases y de los atributos. DAML+OIL permite definir restricciones en atributos de una forma similar a OIL. Un atributo puede definirse como local al especificar donde puede aplicarse y como global cuando no se restringe su aplicación. En casi todos los lenguajes se pueden representar atributos globales excepto en SHOE.

En el concepto de restricciones se visualiza que las restricciones de valor preestablecido son definidas únicamente en XOL, mientras que las de tipo (o intervalo de valores) son incluidas en casi todos los lenguajes. Por otra parte, las restricciones de cardinalidad (valores máximos y mínimos) se incluyen solamente en XOL, OIL y DAML+OIL.

La primitiva subclass-of permite definir taxonomías. En la tabla 1 se observa que todos los lenguajes cuenta con esta primitiva. La descomposición exhaustiva de una subclase sólo se puede representar en OML, OIL y DAML+OIL.

Además, las instancias y los hechos son fácilmente representados en todos los lenguajes. Pero, éste no es el caso de las aseveraciones que sólo pueden representarse en SHOE. XOL y OML no tienen disponibles mecanismos de inferencia, mientras RDF, SHOE, OIL y DAML+OIL si lo tienen y, los servicios de consulta tienen que ser construidos para RDF, SHOE, y DAML+OIL.

2.3.2 Lenguajes de Consulta

El consorcio de la Web estableció el grupo de trabajo XML Query en septiembre de 1999. La misión del grupo es definir un lenguaje estándar y flexible de consulta para extraer datos de documentos reales y virtuales en la Web.

XML-QL

XML-QL [7] tiene un constructor SELECT-WHERE como SQL y toma algunas características de los lenguajes de consulta para la semi-estructura de datos. Además, utiliza elementos XML para especificar restricciones y condiciones. En XML-QL es posible consultar varias fuentes XML simultáneamente para

producir una vista integrada de los datos que cumplen con las condiciones especificadas en la consulta. Los resultados en una consulta se pueden ordenar mediante ORDER BY. Una característica especial de estos lenguajes es que deben permitir traducir datos entre diferentes ontologías, lo que permite escribir consultas que extraigan datos desde bases de datos hacia un DTD y transformarlo en elementos conforme a otra DTD.

Lorel

El lenguaje de consulta Lorel [4] es una extensión de OQL con el estilo de consulta de SQL/OQL. Lorel provee constructores para transformar datos y regresar resultados estructurados. Uno de los constructores es WITH usada con SELECT-WHERE que hace que el resultado de la consulta sea la replicación de todos los datos seleccionados por SELECT con todos los datos alcanzables vía la expresión de ruta en WITH. La expresión de ruta en Lorel puede incluir operadores de expresiones regulares o comodines.

XQL

XQL [7] es un lenguaje conciso desarrollado como una extensión de XSL *pattern language*, conocido como XPath, el cual describe los tipos de nodos a buscar usando un modelo simple de patrón. Este lenguaje no provee condiciones cruzadas para juntar documentos XML con diferentes DTDs, pero provee operadores AND, OR y NOT para consultar documentos con una DTD sencilla. También, tiene el operador * como un carácter comodín para especificar una ruta arbitraria en la consulta. Otra característica de este lenguaje es que puede evaluar cualquier nivel del documento sin tener que navegar desde el elemento raíz. El resultado de una consulta puede ser un nodo, una lista de nodos o cualquier otra estructura.

QUILT

QUIL [4] se desarrolló manteniendo el diseño de un lenguaje de tamaño reducido en el cual sus autores tomaron algunas de las mejores ideas de los

lenguajes de consulta existentes en particular de XML-QL y XQL. Además, introdujeron algunas ideas de SQL y OQL, integrándolas con una nueva sintaxis. El proceso de diseño involucró adaptar características de lenguajes existentes y ensamblarlos en un nuevo diseño. Una consulta simple en QUILT consiste de FOR, WHERE y RETURN. FOR se utiliza para vincular las variables de una o más variables. WHERE aplica un filtro a las variables y RETURN genera un nuevo documento usando los valores seleccionados.

2.4 Programación lógica

La programación lógica inició a principios de los años 70's como una línea de investigación independiente de la demostración automática de teoremas y de la inteligencia artificial. A principios de los 60's tuvo lugar una intensa actividad de investigación por parte de Prawitz, Gilmore, Davis, Putnam y otros en el área de demostración automática de teoremas cuyos esfuerzos culminaron en 1965 con la publicación por parte de Robinson de un artículo [27] en donde introduce los principios de unificación y resolución. El principio de resolución es una regla de inferencia sólida y completa que puede ser eficientemente implementada en computadoras.

La lógica de primer orden había sido usada casi exclusivamente como un lenguaje de especificación o declarativo en las ciencias de la computación. Pero fue hasta 1972 cuando Roussel concibió el acrónimo PROLOG (PROgramming in LOGic) y escribió el primer intérprete en el lenguaje ALGOL-W. El crédito de la introducción de la programación lógica es principalmente para Kowalski y Colmerauer, aunque, Green y Hayes también merecen mencionarse por sus contribuciones. Kowalski y Colmerauer propusieron que la lógica puede usarse como lenguaje de programación. Kowalski mostró que la lógica tiene una interpretación procedural, la cual conduce a implementaciones comparables en eficiencia a aquellas de los lenguajes imperativos como Pascal o C. Una de las principales ideas de la programación lógica se debe a Kowalski que sugiere que en

un algoritmo se pueden distinguir dos componentes disjuntos: la lógica y el control. El componente lógico se refiere a describir el problema a ser resuelto, mientras que el componente de control se refiere al mecanismo que lo resuelve.

2.5 Motores de inferencia de Prolog

Existen varias implementaciones de Prolog en Java. En esta sección sólo se da una breve reseña de algunos de ellos.

2.5.1 Prolog Café

Prolog Café [16] es una herramienta que traduce código Prolog a Java. Además, tiene una interfaz y se puede llamar código Prolog desde programas Java e inversamente. Es un software libre y fue desarrollado por Mutsunori Banbar y Naoyoki Tamura.

2.5.2 GNU Prolog

GNU Prolog [17] es un compilador de Prolog libre, acepta programas Prolog + restricciones y produce código nativo, obteniendo un programa ejecutable. El desempeño de GNU Prolog es muy alentador comparado con los sistemas comerciales. Esta implementación se apega al estándar ISO de Prolog. Además, incluye algunas extensiones muy útiles en el desarrollo de aplicaciones

2.5.3 JavaLOG

JavaLog [18] es interprete de Prolog escrito en Java diseñado para permitir una fácil integración entre Java y Prolog. Fue desarrollado en el instituto de investigación ISISTAN y actualmente se utiliza en varios proyectos de investigación en inteligencia artificial basándose en los conceptos de la programación orientada a objetos para el desarrollo de Software. JavaLog es parte del proyecto Brainstorm, permite la creación de software mezclando los paradigmas de la programación lógica y orientada a objetos, soporta el manejo de conocimiento común de varias instancias para un interprete Prolog, lo que significa que tiene una

arquitectura de pizarra. Permite la distribución física de interpretes de Prolog usando Java RMI y un modulo móvil de lógica.

2.5.4 JProlog

Jprolog [19] es un interprete escrito en Java por Bart Demoen y Paul Taraulog, este sistema cuenta con un recolector de basura y permite traducir programas Prolog a Java y viceversa, pero no cuenta con una interfaz gráfica.

2.5.5 NetProlog

NetProlog [20] es un sistema de programación lógica que genera código binario, ejecutable en la máquina virtual de Java. Sigue casi en su totalidad la sintaxis tradicional del utilizado en las implementaciones de ISO Prolog. Para cada predicado lógico se genera su correspondiente clase en Java. Cuenta con una interfaz gráfica donde se puede editar, compilar e imprimir el código y los objetos.

2.6 Conclusiones

En el transcurso de este capítulo se revisaron los principales conceptos, los cuales son necesarios para la realización del proyecto de tesis con el fin de proporcionar un panorama general de las tecnologías y trabajos relacionados con este trabajo de tesis.

Capítulo 3

Caso de estudio: Deducción de relaciones de parentesco

El propósito de este capítulo es doble: en primer lugar, introducir el lenguaje LogCIN-XML, mostrando el estilo de programación lógica que se puede conseguir, y segundo, presentar el caso de estudio que será usado en esta tesis. La aplicación que se ha desarrollado para demostrar las capacidades deductivas del sistema corresponde al bien conocido problema de parentesco familiar. El problema consiste en definir reglas que permitan deducir cuál es el parentesco que relaciona a dos individuos.

Esta aplicación permite mostrar algunas de las características más sobresalientes del lenguaje como es su capacidad para recuperar información que relaciona a dos o más documentos de acuerdo a un conjunto preciso de condiciones lógicas. Este enfoque contrasta con el seguido en la mayoría de las máquinas de búsqueda tradicionales que agrupan documentos por la similitud en la frecuencia relativa de palabras clave en vez de relacionarlos mediante condiciones lógicas. Al no contar con la capacidad deductiva resolver este problema mediante los buscadores actuales resulta tedioso, ya que se tendrían que definir todas las relaciones de manera explicita. Y más aún, al momento de realizar una consulta no se tiene la capacidad de precisar la información que se desea recuperar y el resultado a nuestra consulta depende en mucho del método de recuperación que se esté utilizando ya que las relaciones de parentesco definidas pasan a ser simples datos.

El lenguaje LogCIN-XML se encuentra formado por elementos XML, términos XML, procedimientos y consultas. A continuación se especifican la gramática abstracta del lenguaje LogCIN-XML.

3.1 Elementos XML

Aunque similares, los elementos XML poseen una estructura más compleja que aquella que poseen los términos en un lenguaje de primer orden debido a la presencia de una lista no ordenada de pares atributo-valor. Los elementos XML consisten del nombre del elemento y de una lista, posiblemente vacía, de atributos ordenados lexicográficamente por el nombre del atributo. La sintaxis de XML establece una distinción entre los tipos texto y cadenas de caracteres. Una constante de tipo texto es una secuencia de caracteres delimitados por elementos, mientras que una constante de tipo cadena de caracteres es una secuencia de caracteres rodeadas por comillas. Por ejemplo, el elemento persona que se muestra a continuación:

```
<persona>
  <nombre sexo="femenino">Karina</nombre>
  <edad>24</edad>
</persona>
```

establece que el elemento *persona* está compuesto a su vez por los elementos nombre y edad, cuyos valores respectivos son *Karina* y *24*, ambos de tipo texto. El elemento nombre posee además una lista de atributos aunque dicha lista consiste únicamente del atributo sexo que tiene el valor "*femenino*" de tipo cadena de caracteres.

La composición de las listas de atributos y de elementos es muy similar a la composición de términos sin variables, lo que permite descomponer un documento en sus piezas básicas de información usando los métodos usuales de programación en lógica.

3.2 Términos XML

Desde el punto de vista de la programación lógica, los elementos XML corresponden a términos sin variables. LogCIN-XML extiende los elementos XML con la noción de variable lógica para introducir términos XML con variables. La importancia de las variables lógicas radica en que permiten definir condiciones que debe cumplir la información deducida por el sistema.

Existen dos tipos de variables en LogCIN-XML: variables de tipo secuencia de caracteres y de tipo término. El tipo se determina a partir del símbolo que precede al nombre de la variable. Las variables cuyo nombre es precedido por el símbolo $ son de tipo texto, mientras que las variables cuyo nombre es precedido por el símbolo # son de tipo término. Las variables anónimas están permitidas en LogCIN-XML y se designan por $_ y #_ de acuerdo a su tipo.

En el ejemplo anterior se introdujo el elemento persona que ahora ha sido reescrito como un término con variables. Las variables $S, $N de tipo texto y #E de tipo elemento se han introducido para establecer condiciones lógicas que las variables deben cumplir.

```
<persona>
  <nombre sexo="$S">$N</nombre>
  #E
</persona>
```

Si los elementos presentados en las secciones precedentes, con y sin variables, fueran sintácticamente idénticos, entonces las variables $S y $N tomaran respectivamente los valores *femenino* y *Karina*, mientras que la variable #E tomara el valor $<edad>24</edad>$.

3.2 Procedimientos

Los procedimientos representan a los axiomas que describen formalmente las características de un sistema, estableciendo las restricciones lógicas que la información debe cumplir. Para teorías de primer orden que usan el lenguaje de las cláusulas de Horn, los axiomas se pueden interpretar como procedimientos. Así, el cuerpo del procedimiento describe la secuencia de acciones cuya realización conduce a obtener la información que cumple las restricciones establecidas por los axiomas del sistema.

Sintácticamente, un procedimiento lógico consiste de una secuencia de invocaciones a otros procedimientos lógicos. En LogCIN-XML se pueden clasificar en procedimientos generadores, predicados primitivos o predicados definidos por el programador. En general, el nombre de un procedimiento tiene ámbito global mientras que el nombre de una variable tiene ámbito local al cuerpo del procedimiento. La única excepción a esta regla son los procedimientos generadores que permiten al ámbito de sus variables extenderse al ámbito del procedimiento que las invoca, como se describe en la sección 3.2.1. Los predicados primitivos son aquellos predefinidos por el sistema, mientras que los procedimientos definidos por el programador establecen una secuencia de acciones.

Entre las características más sobresalientes del modelo de programación lógica de LogCIN-XML se encuentra su capacidad de buscar información sobre colecciones de documentos. El mecanismo que usa para este fin se le conoce como búsqueda con retroceso (*backtracking*) que ha sido ampliamente discutido en la literatura relacionada con el lenguaje Prolog [9]. La búsqueda con retroceso permite obtener toda la información que satisface las condiciones dadas en una consulta, haciendo una búsqueda sistemática que genera el espacio de soluciones factibles. Sin embargo, el problema con dicho espacio de búsqueda es que algunas de las variables pueden tomar sus valores sobre dominios infinitos lo que puede conducir a programas que no terminen. Por otra parte, en espacios de soluciones finitos, la búsqueda con retroceso ha dado lugar a técnicas de programación que permiten

obtener listas que contienen el conjunto completo de soluciones de una consulta. De esta forma, en LogCIN-XML es posible generar el contenido de nuevos documentos a partir de la información extraída sobre colecciones de documentos usando métodos de satisfacción de restricciones.

Los diferentes tipos de procedimientos que se pueden usar en LogCIN-XML se describen con más detalle a continuación.

3.2.1 Procedimientos generadores

El propósito de los procedimientos generadores es el de recuperar información mediante consultas formuladas a colecciones de documentos. Para poder obtener información de un documento, el procedimiento generador debe poseer su misma estructura sintáctica, usando variables que permitan identificar aquellas partes del documento que se desean recuperar.

Por ejemplo, el documento cuyo contenido se muestra a continuación:

<padre-de>
<padre>**fernando**</padre>
<hijo>**victor**</hijo>
</padre-de>

describe una instancia de la relación *padre-de*, en donde **fernando** y **victor** juegan los papeles de *padre* e *hijo*, respectivamente. Para determinar cuál es el nombre del padre de **victor**, podemos usar la variable $P en el siguiente procedimiento generador:

<padre-de>
<padre>$P</padre>
<hijo>**victor**</hijo>
</padre-de>

Al ejecutar este procedimiento obtenemos la solución $P="**fernando**". En forma similar, el siguiente documento que describe una instancia de la relación *esposo-de*,

en donde **fernando** y **susana** desempeñan los papeles de *marido* y *mujer*, respectivamente:

<esposo-de>
<marido>**fernando**</marido>
<mujer>**susana**</mujer>
</esposo-de>

A partir del contenido de este documento podemos deducir que la esposa de **fernando** es **susana**, usando el siguiente procedimiento generador:

<esposo-de>
<marido>**fernando**</marido>
<mujer>$M</mujer>
</esposo-de>

el cual obtendrá la solución $*M*="**susana**".

Los procedimientos generadores pueden compartir variables cuando aparecen en el cuerpo del mismo procedimiento. Por ejemplo, para saber si **susana** es madre (política) de **victor**, podemos formular la siguiente consulta:

<padre-de>
<padre>$P</padre>
<hijo>**pedro**</hijo>
</padre-de>

<esposo-de>
<marido>$P</marido>
<mujer>**susana**</mujer>
</esposo-de>

que relaciona los contenidos de dos documentos (con las relaciones *padre-de* y *esposo-de*) mediante la variable $P cuya solución debe cumplir las condiciones de ser padre de **victor** y esposo de **susana**. El concepto de variable lógica es

fundamental para establecer relaciones entre los contenidos de dos o más documentos. Las variables usadas en los procedimientos generadores extienden su ámbito al cuerpo del procedimiento que las invoca, por lo que se pueden usar para pasar valores entre procedimientos generadores. Modificando las reglas de ámbito para los procedimientos generadores se consigue escribir consultas que resultan más fáciles de entender.

En la Fig. 3 se muestra algunos ejemplos de las relaciones que se pueden definir para el problema de las relaciones de parentesco.

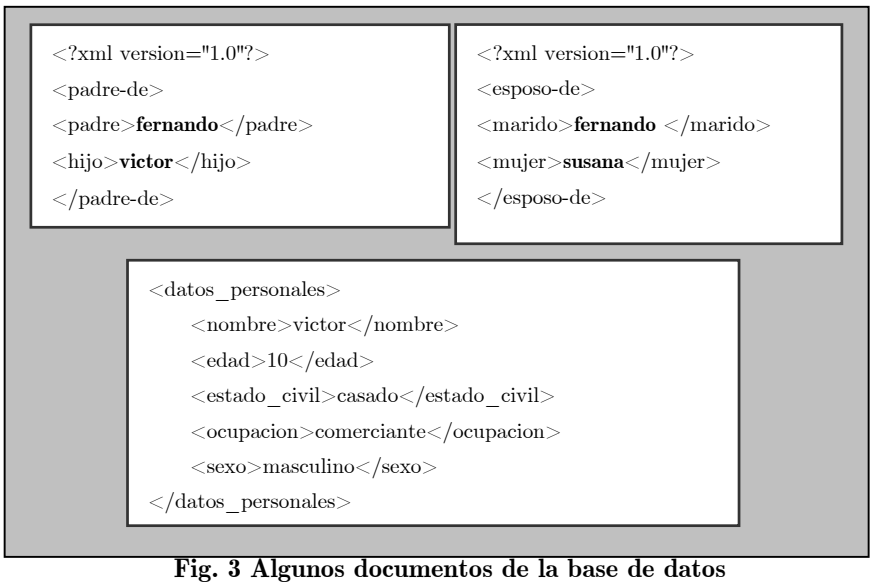

Fig. 3 Algunos documentos de la base de datos

La relación *datos-personales* que aparece en la Fig. 3 se ha introducido con el fin de obtener información adicional sobre los miembros de la familia. Así mismo, esta relación demuestra que el contenido de los documentos puede ser tan completo y tan detallado como sea necesario.

Generalmente, los datos obtenidos deben ser validados lo que puede requerir del uso de un gran número de predicados primitivos los cuales ayudan a reducir el esfuerzo de programación como se explica a continuación.

3.2.2 Predicados primitivos

Los predicados primitivos son aquellos predefinidos por LogCIN-XML. En muchas ocasiones, los predicados primitivos amplían considerablemente el poder del modelo de programación lógica. No obstante, el aumento en el poder expresivo del modelo trae consigo la pérdida en su consistencia y uniformidad al introducir efectos colaterales indeseables.

Los procedimientos primitivos se agrupan por categorías:

Unificación y comparación de términos.

Conjuntos de soluciones.

Meta-predicados.

Modificación de programas.

Aritmética.

Lectura/Escritura de términos.

A diferencia de los procedimientos generadores, los predicados primitivos se invocan siguiendo un estilo que es más parecido al de los lenguajes imperativos como Pascal o SmallTalk. Por ejemplo, para asignar el resultado de la evaluación de una expresión aritmética a una variable, el predicado primitivo *is* se usa de la siguiente forma:

<is variable="$Y" expression="$X+1"/>

Este ejemplo muestra como se asigna a la variable $Y el valor asignado a la variable $X más uno. Aunque el atributo *expression* es de tipo cadena de caracteres, la expresión $X+1 se considera de tipo numérico debido a que dicha expresión ocurre en el contexto del predicado primitivo *is* que establece que su segundo parámetro debe ser de tipo numérico. En general, cada procedimiento primitivo establece el tipo de las expresiones que involucra de modo que sean de tipos compatibles.

Entre los predicados primitivos más importantes se encuentran los meta-predicados *assert* y *retract* para modificar el contenido de los programas. El predicado *assert* introduce un nuevo documento a la colección sin considerar sus posibles repeticiones, mientras que el predicado *retract* elimina un documento de la colección siempre que concuerde con la estructura y contenido del documento dado por el predicado. Por ejemplo, para modificar el conocimiento que se tiene de las relaciones de parentesco para reflejar el hecho que victor tiene un nuevo hijo sam, se usa el predicado *assert* de la siguiente forma:

<assert>

<hijo-de>

<padre>victor</padre>

<hijo>sam</hijo>

</hijo-de>

</assert>

Al ejecutar este predicado, se introducirá el documento

<hijo-de>

<padre>victor</padre>

<hijo>sam</hijo>

</hijo-de>

a la colección de documentos que forman el programa.

3.2.3 Procedimientos definidos por el programador

Los procedimientos definidos por el programador permiten introducir restricciones lógicas más complejas que aquellas que se pueden obtener con los procedimientos generadores o primitivos. Las restricciones se escriben en forma de una secuencia de predicados (primitivos, generadores o definidos por el programador) que se relacionan unos con otros mediante variables lógicas.

Generalmente, los procedimientos se escriben siguiendo el estilo conocido como genera y prueba (*generate and test*) que como sugiere su nombre consiste en generar información mediante consultas a las colecciones de documentos para luego seleccionar aquella que cumple con las restricciones indicadas para finalmente construir un documento con la información procesada de esta forma.

Para el caso de estudio de las relaciones de parentesco, a partir de las relaciones básicas de *padre-de* y *esposo-de* se definieron nuevas relaciones *madre-de*, *hermano-de*, *hijo-de*, *tio-de*, *abuelo-de*, *primo-de*. Por otra parte, los procedimientos *edad-de*, *fecha_de_nacimiento-de*, *ocupación_de* permiten extraer piezas de información de los documentos *datos_personales*.

En LogCIN-XML, la relación *hijo-de* se puede derivar de la relación *padre-de* como se muestra a continuación:

```
<hijo-de padre="$X" hijo="#Y">
   <padre-de>
      <padre>$X</padre>
      #Y
   </padre-de>
</hijo-de>
```

En este ejemplo, la relación *hijo-de* usa las variables $X y #Y como parámetros del procedimiento. A diferencia de la variable $X, la variable #Y debe vincularse a un elemento XML que se pasa como parámetro en forma de texto pero que se interpreta apropiadamente en el contexto del documento XML en el que aparece. Por ejemplo, la invocación:

```
<hijo-de padre="fernando" hijo="$Z"/>
```

obtiene como solución $Z="<hijo>victor</hijo>" cuando se usa la base de documentos de la Fig. 3. La expresión *<hijo>victor</hijo>* es una forma alternativa de escribir el elemento *<hijo>victor</hijo>* en donde las

referencias *<lt;* y *>gt;* evitan usar los caracteres especiales < y >. LogCIN-XML interpreta correctamente este fragmento de texto como un elemento cuando aparece en el contexto apropiado. Mientras este mecanismo permite pasar elementos XML en el paso de parámetros, no permite pasar términos XML con variables ya que, en la actual implementación, las variables no se pueden crear dinámicamente.

Los procedimientos permiten relacionar varios documentos mediante variables lógicas. A continuación se muestra la definición del procedimiento *madre-de* que usa esta característica para determinar si los nombres de dos personas están relacionadas a partir de las relaciones *padre-de* y *esposo-de*.

```
<madre-de madre="$Madre" hijo="$Hijo">
      <padre-de>
            <padre>$Padre</padre>
            <hijo>$Hijo</hijo>
      </padre-de>
      <esposo-de>
        <marido>$Padre</marido>
        <mujer>$Madre</mujer>
      </esposo-de>
</madre-de>
```

Desde el punto de vista de la interpretación declarativa del procedimiento, las variables locales, como la variable $Padre, se interpretan como variables cuantificadas existencialmente, por lo que su ámbito se reduce al cuerpo del procedimiento en donde ocurren. Las variables locales sirven para establecer condiciones sobre el contenido de dos o más documentos.

El cuerpo de un procedimiento se podría interpretar también como una consulta, en donde los parámetros del procedimiento deben fijarse antes de formularla. Las consultas serán presentadas a continuación.

3.4 Consultas

El propósito de una consulta es recuperar la información que se deriva lógicamente tanto de los procedimientos como de la colección de documentos XML. Una vez recuperada la información, ésta se puede usar para construir nuevos documentos. En LogCIN-XML, las consultas son sintácticamente similares al cuerpo de los procedimientos, excepto que el encabezado del procedimiento se reemplaza por el elemento *query*. La máquina de inferencia de LogCIN-XML buscará resolver todas las variables que aparecen en la consulta, produciendo todas las soluciones posibles.

La siguiente consulta busca aquellos hijos que sean menores de 12 años. El elemento *assert* establecen la estructura y contenido del documento que se va a crear. Los elementos que ocurren antes del elemento *assert* son invocaciones a procedimientos que permiten extraer, restringir y procesar la información contenida en la colección de documentos XML.

```
<query>
   <padrede docbase="localhost_8080/logcin/familia">
            <padre>$Y</padre>
            <hijo>$X</hijo>
   </padrede>
   <datos-personales>
      <nombre>$X</nombre>
      <edad>$E</edad>
      #_
      #_
   </datos-personales>
   <lessthan value1="$E" value2="12">
       <assert into="c://logcinserver/salida/ancestro.xml">
            <resultado>
               <persona nombre = "$X"/>
```

```
            <padre nombre = "$Y"/>
            <edad anios = "$E"/>
        </resultado>
    </assert>
</query>
```

Observe el uso de dos variables anónimas #_ que se usan para descartar elementos irrelevantes. Aunque sin uso, estos elementos deben indicarse para asegurar que el documento *datos-personales* posee la estructura correcta. De esta forma, la consulta anterior dará como resultado el siguiente documento:

```
<resultado>
  <persona nombre = "victor"/>
  <padre nombre = "fernando"/>
  <edad anios = "10"/>
</resultado>
```

cuando se usa con la colección de documentos mostrado en la Fig. 3. El documento será insertado con el nombre de archivo *c://logcinserver/salida/ancestro.xml* en el servidor que atendió la consulta y con la ruta indicada.

3.5 Resolviendo consultas en LogCIN-XML

La interpretación de una regla de Horn como procedimiento de la semántica declarativa, como aquella usada en el lenguaje de programación lógica Prolog, permite definir un modelo operacional para LogCIN-XML que es la base de diseño del sistema distribuido en la Web. En la Fig. 4 se muestra la organización general del sistema así como la secuencia de pasos que sigue para responder consultas, en esta figura se utiliza la consulta:

```
<query>
    <primo primo1 = "sonia" primo2="$Y" />
```

</query>

A continuación se describen los pasos indicados en la figura.

Paso 1. En este paso el usuario define la consulta mediante un documento XML utilizando variables para indicar que información se desea recuperar de los documentos. Además, en este momento puede establecer restricciones adicionales a la información que se desea recuperar. Una vez terminado el documento de consulta, esté es enviado al servidor de la aplicación.

Paso 2. Al recibir el documento, el sistema traduce el elemento raíz a un predicado en Prolog y las expresiones XML que contiene a términos en Prolog usando las API's, SAX y DOM. Estas API's analizan, valida y recuperan fragmentos de documentos XML.

Paso 3. La máquina virtual de Prolog inicia el mecanismo de inferencia para verificar si la consulta es consecuencia lógica de los programas P. Para recuperar los documentos XML Prolog se apoya en la máquina virtual de Java.

Paso 4. La máquina virtual de Java recupera los documentos XML que le indican la máquina virtual de Prolog, siguiendo la interpretación procedural en la que cada predicado es un documento.

Paso 5. En este paso la máquina virtual de java le proporciona los documentos solicitados a la máquina virtual de Prolog en forma de términos usando el API en Java que implementa Prolog.

Paso 6. Al recibir la información solicitada, la máquina virtual de Prolog verifica por refutación si la consulta es consecuencia lógica de los programas P; si es así, obtiene la vinculación de variables con los valores que satisfacen la consulta.

Paso 7. Los términos encontrados en la consulta se traduce a un documento XML en donde el predicado en Prolog representa el elemento raíz.

Paso 8. Por último el documento XML generado se visualiza usando XSLT (Hojas de estilo).

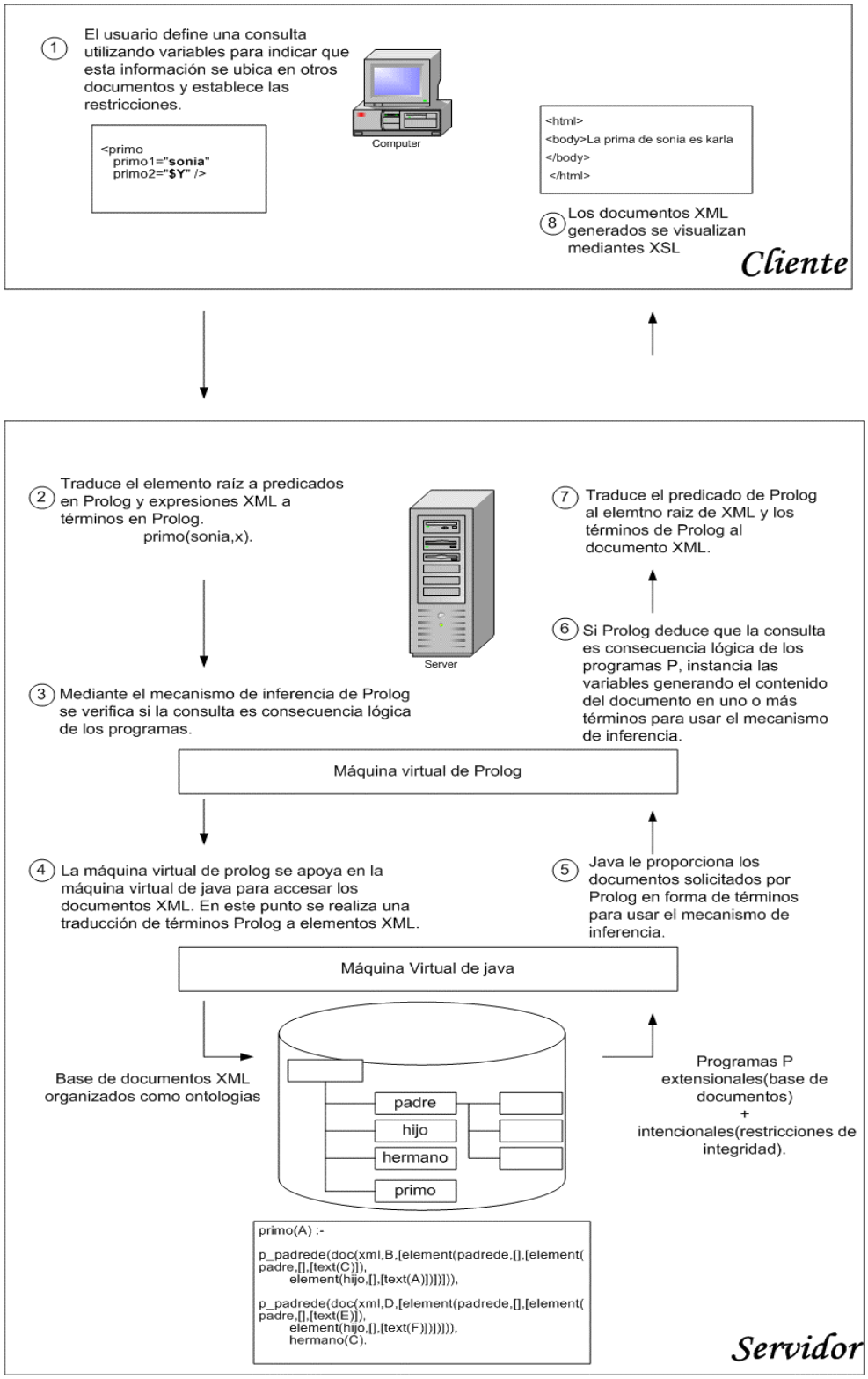

Fig. 4 Esquema general del sistema

Como resultado de este proceso de recuperación de la información se obtiene el siguiente documento XML:

<result>

<primo primo1 = "sonia" primo2="karla" />

</result>

que difiere del documento de la consulta por el elemento raíz result y porque la variable lógica $Y ha sido substituida la constante “karla” que es la respuesta a la consulta.

Este documento al formatearse con XSLT produce el siguiente documento HTML el cual puede visualizarse en cualquier navegador Web.

<html>

<body>La prima de sonia es karla</body>

</html>

Una vez recuperada la información, el usuario puede interrogar al sistema con otras consultas. Como puede constatarse, la información recuperada por LogCIN-XML no podría haber sido recuperado por ninguno de los métodos clásicos de recuperación de información como aquellos en los que se basan los motores de búsqueda de la Web Yahoo, Google, etc.

3.4 Conclusiones

En este capítulo hemos presentado brevemente las características principales del lenguaje LogCIN-XML principalmente con relación a su sintaxis. El significado de las construcciones sintácticas ha sido presentado de manera informal.

Puesto que existen innumerables posibilidades en la definición de la sintaxis de LogCIN-XML, algunos de los criterios principales de este diseño son:

Evitar marcado innecesario. En comparación de otros lenguajes de reglas para la como RuleML, LogCIN-XML no introduce marcado especial (espacio de nombres). La ausencia de marcado especial hace que LogCIN-XML se sitúe a un nivel tan fundamental como aquel en el que se encuentra XML ya que lo único que introduce es una notación para las variables lógicas. Por otra parte, la noción de cláusula (procedimiento lógico) se introduce por la lectura declarativa (operacional) del elemento XML.

Preservar interoperabilidad. Al eliminar el marcado especial en LogCIN-XML se consigue que los documentos generados por terceras aplicaciones puedan leerse o modificarse por ellas. Por ejemplo, RuleML requiere que un documento XML tenga que modificarse para insertar el elemento <fact> como elemento raíz con el fin de que pueda usarse como cláusula unitaria; en contraste, LogCIN-XML no requiere modificar el documento. Esta decisión de diseño supera enormemente las colecciones de documentos que se pueden procesar con LogCIN-XML con relación a aquellas que se pueden procesar con RuleML y otros lenguajes de marcado de reglas.

Colapsar jerarquías de lenguajes. LogCIN-XML no introduce ningún lenguaje mediante elementos de marcado lo que resulta en la imposibilidad de crear una jerarquía de lenguajes. Por ejemplo, en RuleML la clase de los hechos (cláusulas unitarias) es una subclase de las reglas de derivación (cláusulas no unitarias) la cual, a su vez, es una subclase de las reglas.

Permitir intercambiar atributos y subelementos. Mientras que en XML se hace una clara distinción entre atributos y subelementos, en LogCIN-XML dicha distinción es superflua. Para borrar esta distinción, LogCIN-XML por una parte introduce elementos (cuyo marcado se codifica con referencias) en los valores de los atributos y fija el orden de los atributos conforme a la declaración del elemento.

Revisar su compatibilidad con DTDs y Schemas. Debido al nivel tan fundamental en el que se sitúa LogCIN-XML, parece inevitable revisar su compatibilidad con las actuales especificaciones de DTDs y Schemas.

Eliminar marcado para conjunciones. En vez de dejar la conjunción con un marcado explícito como <and>...</and> para agrupar una secuencia de conyuntos, se ha preferido hacer un marcado implícito entre ellos.

Permitir modelo unificado de datos. Como resultado de discusiones previas, en LogCIN-XML es posible usar el modelo unificado de datos XML-RDF con árboles ordenados etiquetados como base de la notación.

Permitir definición de roles (papeles). Por conveniencia se introduce marcado para predicados similares a aquellos de RDF, llamados aquí roles, con marcas que usan '_' como prefijo. La principal ventaja de los roles es que están orientados a la modelación con objetos. Por ejemplo, si alguna información adicional se va a incluir en un elemento, entonces es más fácil hacerlo al estilo de RDF agregando un nuevo triple que relacione al elemento con el nombre del atributo y su valor que al estilo de XML agregando un nuevo elemento a la secuencia de descendientes. Manteniendo separado la adición de nueva información se consigue facilitar la lectura, reducir o eliminar las modificaciones en las posiciones de los elementos y preservar aquellas aplicaciones que usan los datos (por ejemplo, aquellas que usan procesamiento vía XSLT).

Usar localizadores universales de recursos. Los identificadores universales de recursos se usan para localizar, describir y acceder a recursos y servicios tales como clases, objetos, agentes de software, componentes Web, servicios Web, etc. La representación de objetos como URIs en RuleML facilitará la integración con el trabajo relacionado con ontologías. Los objetos y servicios Web usan URL/URI como su único identificador de objeto (como en SHOE, RDF y URML) así como su punto de acceso a un recurso Web o agente de software. Los URL/URI pueden usarse en la descripción de hechos, condiciones y acciones de reglas. Los URIs pueden designar, por ejemplo, tanto a individuos como a relaciones y también pueden marcar el contenido de un elemento.

A pesar de la sencillez del caso de estudio se puede visualizar las capacidades del sistema LogCIN-XML. En este capítulo sólo se da un panorama general de los

elementos que forman el caso de estudio para que el lector se familiarice con ellos ya que los ejemplo que se encuentra en los siguientes capítulos están dados apoyándose en este caso de estudio. En el siguiente capítulo se trata a detalle la sintaxis y semántica de LogCIN-XML.

Capítulo 4

Formalización del lenguaje LogCIN-XML

En este capítulo se describe la formalización del lenguaje LogCIN-XML. LogCIN-XML es un lenguaje de primer orden que ha sido definido siguiendo el modelo de programación lógica de Prolog. De acuerdo con la interpretación de Kowalski, el componente lógico de un programa se puede entender como una teoría de primer orden en la cual las consultas son afirmaciones cuyas soluciones son consecuencias lógicas de los axiomas de la teoría. Por otra parte, el componente operacional se refiere a la interpretación de los axiomas como la secuencia de acciones que se deben realizar para derivar una consecuencia lógica. Esta relación con las lógicas de primer orden establece el sistema formal que da fundamento al modelo de programación de LogCIN-XML.

Una teoría de primer orden consiste de un alfabeto, un lenguaje de primer orden, un conjunto de axiomas y un conjunto de reglas de inferencia [8]. El lenguaje de primer orden consiste de fórmulas bien formadas que se obtienen a partir de las reglas de composición de la sintaxis abstracta del lenguaje de cláusulas de Horn. Los axiomas de la teoría son fórmulas bien formadas y consisten de cláusulas unitarias y no unitarias. Las reglas de inferencia, fundamentada en los principios de unificación y resolución SLD, usan los axiomas para derivar los teoremas de la teoría.

En este capítulo comenzaremos definiendo la sintaxis de LogCIN-XML para concluir definiendo su semántica declarativa y operacional.

4.1 Sintaxis

La sintaxis de los lenguajes lógicos de primer orden descansan en la noción primordial de *nombre simbólico* el cual permite declarar, categorizar y sugerir una primera interpretación de los datos en forma abstracta de la aplicación. Los nombres simbólicos son elementos lexicográficos que inducen una estructura de términos. La importancia de la estructura de términos es que ofrece una representación universal para cualquier tipo de dato.

4.1.1 Entidades lexicográficas

Los elementos lexicográficos del lenguaje LogCIN-XML consisten de un conjunto de símbolos los cuales se agrupan en clases dentro de un alfabeto.

Definición. Un *alfabeto* está formado por siete clases de entidades lexicográficas:

- Variables
- Constantes
- Símbolos de función
- Símbolos de predicado
- Conectivos
- Cuantificadores
- Símbolos de puntuación

Estas clases de entidades lexicográficas se definen a continuación.

Variables. En LogCIN-XML, las variables representan elementos no determinados pero que pueden tomar sus valores de un dominio dado. Las variables pueden ser de tipo cadena o de tipo término. Las variables anónimas están permitidas y se designan por la subraya '_'. Las variables de tipo cadena, cuyos nombres forman el conjunto XMLStringVariable, son precedidas por el signo '$'. Las

variables de tipo término, las cuales forman el conjunto XMLTermVariable, son prefijadas por el signo '#'. En general, el conjunto de nombres de variables se designa como XMLVariable y corresponde a la unión de XMLTermVariable y XMLTermVariable.

Ejemplos:

XMLStringVariable: $X, $Num, $s, $_.

XMLTermVariable: #T, #Nodo, #otro, #_.

Constantes. Las constantes representan elementos bien determinados los cuales, en conjunto, forman un dominio. LogCIN-XML hace una distinción entre texto el cual es una secuencia de caracteres cuyos elementos forman el dominio XMLText y cadena de caracteres que es una secuencia de caracteres rodeadas por comillas cuyos elementos forman el dominio XMLString. La unión de ambos dominios XMLText y XMLString corresponde genéricamente al dominio de las constantes XMLConstant.

Ejemplos:

XMLText: Esto es texto

XMLString: "Cadena de caracteres"

Símbolos de Atributo. Dado que un atributo es un par nombre-valor de la forma n=v, un símbolo de atributo corresponde con el nombre del atributo n, el valor v de un atributo es una constante (cadena).

Símbolos de Función. Los símbolos de función corresponden con nombres de etiquetas XML (XMLTag). Los símbolos de función tienen asociados una lista de pares atributo-valor y una lista de elementos. Ambas listas pueden ser vacías.

Símbolos de Predicado. Los símbolos de predicado son aquellos definidos por el usuario (Procedure) y aquellos predefinidos en la máquina virtual de Prolog (PredefinePredicate). Al igual que los símbolos de función, los símbolos de

predicado tienen asociados una lista de pares atributo-valor y una lista de elementos. Ambas listas pueden ser vacías.

Conectivos. Los dos conectivos usados en el lenguaje de cláusulas, conjunción e implicación, aparecen implícitos sin ambigüedad en las cláusulas de la teoría. En LogCIN-XML la posición de los conectivos está completamente determinada por lo que se puede omitir su uso. En la sección 4.1.4 se discute el uso implícito de los conectivos.

Cuantificadores. Al igual que los conectivos, los cuantificadores universal y existencial aparecen implícitos sin ambigüedad en las cláusulas de la teoría. En la sección 4.1.4 se discute el uso implícito de los cuantificadores.

Signos de puntuación. Los signos de puntuación de LogCIN-XML son aquellos de XML. Aparte de ellos, LogCIN-XML usa los mismos mecanismos lexicográficos y sintácticos de XML para describir gramáticas abstractas: la identificación y posición de las entidades sintácticas está completamente determinada por la posición que ocupa en el árbol de sintaxis multicamino, ordenado y etiquetado que representa a un elemento XML.

4.1.2 Elementos XML

En relación con el lenguaje de cláusulas, los elementos XML corresponden con términos sin variables (*ground terms*). Sin embargo, aunque similares, los elementos XML poseen una estructura más compleja que la de los términos en un lenguaje de primer orden debido a la presencia de la lista no ordenada de pares atributo-valor asociada al símbolo de función o de predicado.

Definición. Una *lista de atributos* es una secuencia posiblemente vacía de atributos.

En LogCIN-XML el orden en el que aparecen los atributos en una lista es irrelevante. Por tal razón, estableceremos el siguiente convenio de notación en el cual se adopta una representación normalizada de las listas de atributos.

Convenio. Los elementos de una lista de atributos están ordenados lexicográfico, en orden creciente, por el nombre del atributo.

Definición. Un *término sin variables* es un elemento XML que pertenece a la categoría sintáctica XMLElement la cual se define inductivamente a continuación:

Una constante c es un término sin variables.

1. Si A es un símbolo de función y attributelist es una lista de atributos, entonces <A attributelist /> es un término sin variables.
2. Si A es un símbolo de función, attributelist es una lista de atributos y E_1 ,..., E_n son términos sin variables, entonces <A attributelist> E_1 ,..., E_n </A> es un término sin variables.

4.1.3 Términos

En comparación con la gramática de XMLElement, la gramática de los términos XML, la cual se define a continuación, incluye la ocurrencia de variables tanto en la lista de atributos como en los elementos.

Definición. Un *término con variables* el cual pertenece a la categoría sintáctica XMLTerm es definido inductivamente a continuación:

1. Una variable x es un término con variables.
2. Si A es un símbolo de función y a_1=v_1, ..., a_n=v_n (con $n \geq 0$) es una lista de atributos donde el valor de algún atributo es una variable entonces <A a_1=v_1, ..., a_n=v_n /> es un término con variables.
3. Si A es un símbolo de función, a_1=v_1, ..., a_n=v_n (con $n \geq 0$) es una lista de atributos, E_1, ..., E_m son términos con o sin variables y al menos existe una variable en a_1=v_1,..., a_n=v_n o E_1, ..., E_n entonces <A a_1=v_1, ..., a_n=v_n> E_1, ..., E_n </A> es un término con variables.

Tanto los elementos como los términos admiten una representación basada puramente en secuencias de caracteres usando referencias como entidades lexicográficas que son interpretadas en el contexto en el que ocurren. Por ejemplo, el elemento XML <color>azul</color> se puede escribir como < color > azul

< / color > en donde las referencias < y > representan a los caracteres < y > , respectivamente. Esta facilidad, permitirá pasar elementos como parámetros en un paso de inferencia (invocación de un procedimiento).

4.1.4 Cláusulas

En un lenguaje de cláusulas de Horn, una cláusula se compone de un antecedente y un consecuente, separados por el conectivo de implicación. A su vez, el consecuente esta formado por una secuencia de conjuntos separados por el conectivo de conjunción.

Definición. Una *fórmula atómica* o *átomo* <P a_1=v_1,..., a_n=v_n /> consiste de un símbolo de predicado P y una lista de atributos a_1=v_1,..., a_n=v_n (con $n \geq 0$), en donde los a_1,.., a_n son símbolos de atributo y v_1,..,v_n son elementos o términos.

Los símbolos de predicado pueden ser predefinidos o definidos en la especificación del programa. En la lista de atributos, el atributo especial docbase = URL, indica donde se encuentra su definición mediante un localizador uniforme de recursos o URL por sus siglas en inglés (*Universal Resource Locator*). Esta característica será discutida en el capítulo 5.

Definición. Una *literal* es un átomo o un átomo negado sin variables.

En LogCIN-XML la posición del consecuente y de los antecedentes está completamente determinada por lo que se puede omitir el uso de los conectivos: el antecedente se representa por el encabezado de un elemento XML (o predicado LogCIN-XML), mientras que el consecuente se representa por la secuencia de subelementos XML (o predicados LogCIN-XML) que contiene.

Definición. Una *cláusula definida* <P a_1=v_1, ..., a_n=v_n> P_1 ... P_m </P>, perteneciente a la categoría XMLProcedure, consiste de un símbolo de predicado P, una lista de atributos a_1=v_1, ..., a_n=v_n (con $n \geq 0$), en donde los a_1, ..., a_n son símbolos de atributo y v_1, ..., v_n son elementos o términos y una secuencia P_1, ..., P_m (con $m \geq 0$) posiblemente vacía de átomos. A la expresión <P a_1=v_1, ..., a_n=v_n> se le llama *antecedente o encabezado de la cláusula*, en tanto

que a la secuencia $P_1 \ldots P_m$ se le llama *consecuente o cuerpo de la cláusula*. A una cláusula con consecuente vacío se llama *cláusula unitaria*; de otro modo se llama *cláusula no unitaria*.

Definición. En la cláusula definida <P a_1=v_1, ..., a_n=v_n> $P_1 \ldots P_m$ </P> (con $m,n \geq 0$), las variables que aparecen en la lista de atributos del antecedente a_1=v_1, ..., a_n=v_n están *cuantificadas universalmente*, mientras que aquellas que solamente aparecen en el consecuente $P_1 \ldots P_m$ están *cuantificadas existencialmente*.

Definición. El *ámbito* de una variable cuantificada en una cláusula corresponde con el antecedente o cuerpo de la cláusula. Una variable libre es aquella que no está cuantificada (ni universal ni existencialmente).

Definición. Una *fórmula cerrada* es una fórmula que no tiene ocurrencias de variables libres.

Definición. Un *programa* Π, perteneciente a la categoría XMLProgram, es un conjunto finito de cláusulas definidas.

Defínción. En un programa Π, el conjunto de todos los procedimientos con el mismo símbolo de predicado P en el consecuente es llamado la *definición de* P.

Definición. Una *consulta definida* o *meta* <query> $P_1 \ldots P_n$ </query>, perteneciente a la categoría XMLQuery, es una conjunción de literales $P_1,\ldots, P_n$ que pueden incluir variables cuantificadas existencialmente. A cada P_i ($i = 1, \ldots, n$) se llama *submeta*.

Definición. La cláusula *vacía* se denota por empty no posee ni antecedente ni consecuente y denota una *contradicción*.

Definición. Una *cláusula de Horn* es una cláusula o una meta.

Definición. El *lenguaje de cláusulas de Horn* dado por un alfabeto consiste del conjunto de todas las fórmulas bien formadas que se pueden derivar a partir de los símbolos del alfabeto.

4.2 Unificación

En esta sección se introducen conceptos fundamentales de la programación lógica para probar la equivalencia sintáctica de términos como substituciones y unificadores así como el algoritmo de unificación. Al igual que en los procedimientos, el paso de parámetro en las reglas se hace mediante las variables que aparecen en la lista de atributos. Para este fin, el algoritmo de unificación se modifica para comparar los términos con igual nombre de atributo en lugar de comparar los pares de términos que corresponden en su posición.

Dado el usual ordenamiento lexicográfico en nombres ≤ para todo XMLTerm

<A a_1=T_1 a_2=T_2 . . . a_m=T_m> ...</A>

se cumple que: $a_i \leq a_j$ si $i \leq j$ para todo i, j en 1,2,...,m.

4.2.1 Substitución

Definición. Una *sustitución* es una función parcial de variables a términos XML con variables:

σ : XMLVariable → XMLTerm.

El dominio de las substituciones se extiende naturalmente de las variables a los términos.

σ : XMLTerm → XMLTerm.

Definición. A la aplicación de una sustitución σ a un término x se le llama *la instancia del término bajo la sustitución* y se denota por xσ.

Definición. Una *sustitución* se define inductivamente como se muestra a continuación:

1. Si x ∈ XMLString xσ = x
2. Si x ∈ XMLVariable xσ = σ(x)
3. <A a_1=T_1 a_2=T_2 … a_m=T_m/>σ = <A a_1=T_1σ a_2=T_2σ … a_m=T_mσ />

4. <A a_1=T_1 a_2=T_2 … a_m=T_m>… x …</A> σ = <A a_1=T_1σ a_2=T_2σ … a_m=T_mσ>… xσ …</A>

Definición. Una *sustitución directa* es una sustitución a elementos XML ya que no introducen variables.

σ : XMLTerm → XMLElement

Definición. Una *composición de substituciones* se define como se muestra a continuación.

$\sigma_1\sigma_2$ (x) = σ_1(x)σ_2 si x ∈ dom(σ_1) ∧ x ≠ σ_1(x)σ_2

$\sigma_1\sigma_2$ (x) = σ_2(x) si x ∉ dom(σ_1) ∧ x ∈ dom(σ_2)

donde σ(x) denota el término asociado a la variable x en la sustitución σ y dom(σ) denota el conjunto de variables para las cuales la sustitución está definida.

Definición. La *identidad* ε es la sustitución nula para la composición de substituciones εσ = σε = σ.

Proposición. Sean θ, σ y γ sustituciones. Entonces

1. θε = εθ = θ
2. (<A a_1=T_1 a_2=T_2 … a_m=T_m/> θ) σ = <A a_1=T_1 a_2=T_2 … a_m=T_m/> (θσ)
3. (<A a_1=T_1 a_2=T_2 … a_m=T_m>… *x* …</A> θ) σ = <A a_1=T_1 a_2=T_2 … a_m=T_m>… *x*σ …</A>(θσ)
4. (θσ)γ = θ(σγ)

Definición. Sea E un término, literal ó conjunción de literales. Una *sustitución de renombramiento* ocurre si {x_1,...,x_n} es el conjunto de variables que aparecen E y éstas son sustituidas por {y_1, ...,y_n} respectivamente. Además, las y_i son variables todas ellas diferentes y ({x_1,...,x_n}) ∩ {y_1,..., y_n}=∅.

4.2.2 Unificadores

Definición. Un *unificador* es una sustitución que hace a las instancias correspondientes de dos términos x y x' sintácticamente idénticas $x\sigma = x'\sigma$.

Definición. El *unificador más general (mgu)* por sus siglas en ingles es aquel que no puede obtenerse por composición de otro distinto.

$mgu(A,B) = \sigma$ si $(\{A = B\}, \varepsilon) \Rightarrow^* (\varnothing, \sigma)$

En donde la relación $\Rightarrow^*$ es la cerradura reflexiva y transitiva de $\Rightarrow$.

4.2.3 Algoritmo de unificación

El algoritmo de unificación está definido sobre términos XML normalizados. El algoritmo, esencialmente es el mismo que el definido por Martelli-Montanari [10], termina encontrando el unificador más general si éste existe o bien reportando fallo. La unificación de términos XML provee un mecanismo uniforme para el paso de parámetro, la selección, la recuperación y la construcción. A continuación se muestra el algoritmo de unificación.

U1. $(C \cup \{<A\ a_1=S_1\ \ldots\ a_m=S_m> E_1 \ldots E_n </A> = <A\ a_1=T_1\ \ldots\ a_m=T_m> F_1 \ldots F_n </A>\}, \sigma) \Rightarrow (C \cup \{S_1=T_1, \ldots, S_m=T_m, E_1=F_1, \ldots, E_n=F_n\}, \sigma)$

U2. $(C \cup \{T=T\}, \sigma) \Rightarrow (C, \sigma)$

U3. $(C \cup \{T=x\}, \sigma) \Rightarrow (C \cup \{x=T\}, \sigma)$

U4. $(C \cup \{x=T\}, \sigma) \Rightarrow (C\sigma, \sigma\{x \rightarrow T\})$ si $x \notin vars(T)$

U5. fail (fallo) en cualquier otro caso

La regla U1, llamada *descomposición*, reduce la complejidad del algoritmo de unificación de dos términos produciendo un conjunto de ecuaciones que resultan de la comparación de los subtérminos correspondientes. La regla U2, llamada eliminación de *ecuaciones triviales*, elimina ecuaciones que comparan términos idénticos. La regla U3, llamada de *orientación*, deja la variable del lado izquierdo de una ecuación. La regla U4, llamada *eliminación de variables*, substituye la ocurrencia de una variable en el conjunto de ecuaciones y obtiene

4.3 Interpretaciones y Modelos

La semántica declarativa de un programa lógico es dada por la semántica de las fórmulas en la lógica de primer orden. En esta sección se discute brevemente las interpretaciones y modelos.

Definición. Una *pre-interpretación* en LogCIN-XML consiste de lo siguiente:

1. Un conjunto D no vacío el cual representa el contenido de los documentos XML, llamado el *dominio de la pre-interpretación*.
2. A cada constante en LogCIN-XML le corresponde un elemento en D.
3. A cada símbolo de función (nombre de etiqueta) en LogCIN-XML con una lista de n atributos y m subtérminos le corresponde un mapeo desde D^{n+m} a D.

Notación. Usaremos la notación D^{n+m} para designar al dominio de un término LogCIN-XML con n atributos y m subtérminos.

Definición. Sea J una pre-interpretación de LogCIN-XML. Una *asignación de variable* ($\mathsf{wrt\ J}$) es la asignación a cada variable de LogCIN-XML de un elemento en el domino de J.

Definición. Una *interpretación* I de LogCIN-XML consiste de una pre-interpretación J con dominio D de LogCIN-XML tal que para cada símbolo de predicado en LogCIN-XML con una lista de n atributos y m argumentos le corresponde un mapeo de D^{n+m} a {verdadero, falso}. Se dice que I *se basa en* J.

Definición. Sea I una interpretación para LogCIN-XML y sea W una fórmula en LogCIN-XML.

1. Se dice que W es *satisfacible* en I si existe un elemento $d \in \mathsf{D}^{n+m}$ que hace a la fórmula W verdadera con respecto a I.
2. Se dice que W es *válida* en I si todo elemento $d \in \mathsf{D}^{n+m}$ hace a la fórmula W verdadera con respecto a I.

3. Se dice que W es *insatisfacible* en I si existe un elemento $d \in \mathsf{D}^{n+m}$ que hace a la fórmula W falsa con respecto a I.
4. Se dice que W es *inválida* en I si todo elemento $d \in \mathsf{D}^{n+m}$ hace a la fórmula W falsa con respecto a I.

Definición. Sea I una interpretación en LogCIN-XML y sea F una fórmula cerrada de LogCIN-XML. Entones I es un *modelo* para F si F es verdadera con respecto a I.

Definición. Sea T una teoría de primer orden y LogCIN-XML el lenguaje de T. Un modelo para T es una interpretación para LogCIN-XML, la cual es un modelo para cada axioma de T. Si T tiene un modelo, se dice que T es *consistente*.

Definición. Sea S un conjunto de fórmulas cerradas e I una interpretación de LogCIN-XML. Se dice que I es un *modelo* para S, si I es un modelo para cada fórmula de S.

Definición. Sea S un conjunto de fórmulas cerradas de LogCIN-XML.

1. Se dice que S es *satisfacible*, si existe una interpretación la cual es un modelo para S.
2. Se dice que S es *válida* si cada interpretación es un modelo para S.
3. Se dice que S es *insatisfacible* si no existe interpretación que sea un modelo para S.
4. Se dice que S es *inválida* si existe una interpretación que no sea un modelo para S.

A continuación introducimos el concepto fundamental de consecuencia lógica.

Definición. Sea S un conjunto de fórmulas cerradas y F una fórmula cerrada. Decimos que F es *consecuencia lógica* de S si, para interpretación I, I es un modelo de S implica que I es un modelo de F.

Cuando $\mathsf{S} = \{\mathsf{F}_1,...,\mathsf{F}_n\}$ es un conjunto finito de fórmulas cerradas, entonces F es una consecuencia lógica de S si y solo si $\mathsf{F}_1 \wedge ... \wedge \mathsf{F}_n \rightarrow \mathsf{F}$ es válida.

Proposición. Sea S un conjunto de fórmulas cerradas y F una fórmula cerrada. Entonces F es una consecuencia lógica de S si y solo si $\mathsf{S} \cup \{\neg\mathsf{F}\}$ es insatisfacible.

Demostración. Supongamos que F es consecuencia lógica de S. Sea I una interpretación de S que es un modelo para S. Entonces I es un modelo para F. Por lo tanto, I es una interpretación que no es un modelo para $\mathsf{S} \cup \{\neg\mathsf{F}\}$. Puesto que no hay otras interpretaciones que sean modelos para S, se concluye que no hay modelos para $\mathsf{S} \cup \{\neg\mathsf{F}\}$ y de aquí que sea insatisfacible.

Recíprocamente, supongamos que $\mathsf{S} \cup \{\neg\mathsf{F}\}$ es instatisfacible. Sea I una interpretación que es un modelo para S pero que no lo es para $\neg\mathsf{F}$. Entonces, I es un modelo para F. Por lo tanto, F es consecuencia lógica de S. ∇

La importancia de la proposición anterior radica en que muestra que para probar que una consulta G es consecuencia lógica de un programa P, entonces hay que probar que $\mathsf{P} \cup \{\neg\mathsf{G}\}$ es insatifacible. Pero de acuerdo a la definición de insatisfacibilidad, esto requeriría demostrar que cualquier interpretación no es un modelo para $\mathsf{P} \cup \{\neg\mathsf{G}\}$, lo cual es difícil de probar en la práctica, particularmente para dominios infinitos no numerables. Afortunadamente, es posible restringir la demostración de insatisfacibilidad a una interpretación más conveniente, llamada la *interpretación de Herbrand*. La interpretación de Herbrand reduce cualquier interpretación a una en un dominio simbólico apropiado para automatizarse por computadora.

Por abuso de lenguaje, conviene referirse a una interpretación de un conjunto de fórmulas S en lugar del lenguaje de primer orden subyacente. De esta forma, nos referiremos al conjunto de fórmulas que constituyen un programa P, de modo que supondremos que el alfabeto del lenguaje de primer orden está formado por los símbolos de constantes, funciones y predicados que aparecen en el programa.

Definición. El *universo de Herbrand* denotado por U_P para el programa P es el conjunto de todos los términos base que pueden formarse únicamente con las constantes y los símbolos de función que aparecen en P.

Ejemplo. Dado el programa P definido por las siguientes dos cláusulas:

```
<p a="$x">
   <q>
     <f>$x</f>
     <g>$x</g>
   </q>
</p>
<r b="$x"/>
```

Usando los símbolos f y g junto con un elemento adicional c, el universo de Herbrand U_P se obtiene por la composición de estos símbolos:

U_P = {c, <f>c</f>, <g>c</g>, <f><f>c</f></f>, <f><g>c</g></f>,
<g><f>c </f></g>, <g><g>c</g></g>,...}

Definición. La base de Herbrand B_P para P es el conjunto de todos los átomos base que pueden formarse únicamente por composición de los símbolos de predicado con los términos base del universo de Herbrand U_P para P como argumentos.

Ejemplo. Para el ejemplo anterior, la base de Herbrand B_P para P usa los símbolos de predicado p, q y r de modo que:

B_P = { c , <q>c c</q>, <r b="c"/>,
<f>c</f> ,
c , <f>c</f> ,
c , <g>c</g> ,
<f>c</f> ,

<g>c</g> ,

<g>c</g> ,

<f>c</f> ,

<q><f>c</f> c</q>, <q>c <f>c</f></q>,

<q><g>c</g> c</q>, <q>c <g>c</g></q>,

<q><f>c</f> <f>c</f></q>,

<q><g>c</g> <g>c</g></q>,

<q><f>c</f> <g>c</g></q>,

<q><g>c</g> <f>c</f></q>,

<r b="< f > c < /f >"/>,

<r b="< g > c < /g >"/>, ...}

Obsérvese que términos base como **<f>c</f>** también aparecen en la lista de atributos pero escritos en forma equivalente como **< f > c < /f >**.

Definición. La *pre-interpretación de Herbrand* para P es la pre-interpretación dada por lo siguiente:

1. El dominio de la pre-interpretación es el Universo de Herbrand U_P.
2. Las constantes en P se asignan a ellas mismas en U_P.
3. Si A es un símbolo de función de n atributos y m subelementos en P, entonces el mapeo de $(U_P)^{n+m}$ a U_P definido por a_1=S_1 … a_n=S_n → <A a_1=S_1 … a_n=S_n>E_1...E_m</A> se asigna a A.

Definición. Una *interpretación de Herbrand* para P es cualquier interpretación basada en la pre-interpretación para P.

Para cualquier interpretación de Herbrand, la base de Herbrand es el conjunto de todos los átomos base que hacen verdadera a la interpretación. Recíprocamente, cualquier subconjunto de la base de Herbrand se puede considerar como una

interpretación ya que el subconjunto define la asignación del valor verdadero a los elementos del subconjunto. La base de Herbrand es el principio para definir una interpretación que sea un modelo para un conjunto de fórmulas cerradas.

Para probar la insatisfacibilidad de un conjunto de fórmulas cerradas es suficiente considerar modelos de Herbrand siempre que las fórmulas del conjunto sean cláusulas.

Definición. Sea S un conjunto cerrado de cláusulas de P. Un *modelo de Herbrand* para S es una interpretación de Herbrand para P la cual es un modelo para S.

Proposición. Sea S un conjunto de cláusulas. S tiene un modelo si y solo si S tiene un *modelo de Herbrand*.

Demostración. Supongamos que I es una interpretación que es un modelo para S. Entonces, el subconjunto H de la base de Herbrand definido por:

$H = \{<P\ a_1{=}S_1 \ldots a_n{=}S_n{>}E_1 \ldots E_m{<}/P{>} \in B_S$ tal que

$<P\ a_1{=}S_1 \ldots a_n{=}S_n{>}E_1 \ldots E_m{<}/P{>}$ es verdadera en I$\}$.

Por lo tanto, H es un modelo de Herbrand para S.∇

Proposición. Sea S un conjunto de cláusulas. S es insatisfacible si y solo si no tiene modelos de Herbrand.

Demostración. La proposición es lógicamente equivalente a la siguiente: S tiene un modelo de Herbrand si y solo si S es satisfacible, la cual es inmediata ya que si S es satisfacible, entonces tiene un modelo y por lo tanto un modelo de Herbrand. ∇

4.4 Semántica

En esta sección se presenta la semántica declarativa y operacional de un programa definido. Se define la respuesta computada, la solidez y completitud de la resolución-SLD. Además, se introduce el procedimiento de refutación-SLD.

4.4.1 Semántica declarativa

En esta sección se introduce el modelo mínimo de Herbrand de un programa definido. Este modelo en particular juega un papel central en la teoría.

Proposición (Propiedad de intersección de modelos). Sea P un programa definido y $\{\mathsf{M_i}\}_{\mathsf{i}\in\mathsf{I}}$ sea un conjunto no vacío de modelos de Herbrand para P. Entonces $\cap_{\mathsf{i}\in\mathsf{I}}\mathsf{M_i}$ es un modelo de Herbrand para P.

Definición. Dado un programa definido P, a la intersección de todos los modelos de Herbrand se le llama el *modelo mínimo de Herbrand* y se denota por $\mathsf{M_P}$.

Teorema. Sea P un programa definido. Entonces $\mathsf{M_P} = \{\mathsf{A}\in\mathsf{B_P} : \mathsf{A}$ es consecuencia lógica de $\mathsf{P}\}$.

Demostración.

A es consecuencia lógica de P.

$\mathsf{P} \cup \{\neg\mathsf{A}\}$ es insatisfacible.

Para toda interpretación $\mathsf{P} \cup \{\neg\mathsf{A}\}$ no tiene modelos de Herbrand.

$\neg\mathsf{A}$ es falsa con respecto a todos los modelos de Herbrand de P.

A es verdadera con respecto a todos los modelos de Herbrand de P.

$A \in M_P$.

∇

Caracterización de punto fijo

El modelo mínimo de Herbrand induce una estructura de retículo por lo que se puede caracterizar como el punto fijo de un proceso iterativo.

Definición. Sea P un programa definido. El mapeo $\mathsf{T_P}$: $2^{\mathsf{B}}{}_{\mathsf{P}} \rightarrow 2^{\mathsf{B}}{}_{\mathsf{P}}$ se define como sigue. Sea I una interpretación de Herbrand. Entonces $\mathsf{T_P(I)} = \{$<A/>$\in\mathsf{B_P}$: <A><$\mathsf{A_1}$/>...<$\mathsf{A_n}$/></A>$\}$ es una instancia base de una cláusula en P y $\{$<$\mathsf{A_1}$/>,..., <$\mathsf{A_n}$/>$\} \subseteq \mathsf{I}$.

Definición. Sea S un conjunto y 2^S el conjunto de todos los subconjuntos de S. 2^S es un *conjunto parcialmente ordenado* bajo la relación de subconjuntos $\subseteq$. Entonces, $a \in S$ es una *cota superior* de un subconjunto X de S si $x \subseteq a$, para toda $x \in X$. Similarmente, $b \in S$ es una *cota inferior* de X si $b \subseteq x$, para toda $x \in X$.

Sea P un programa definido. Entonces, el conjunto $2^{\mathsf{B}_\mathsf{P}}$ de todas las interpretaciones de Herbrand de P es un retículo completo bajo la relación $\subseteq$ de orden parcial de inclusión de conjuntos. En el retículo, el elemento tope es B_P, mientras que el elemento base es $\varnothing$.

Definición. Un conjunto parcialmente ordenado L se dice que es un *retículo completo* si existen una *cota superior mínima* lub (por sus siglas en inglés *least upper bound*) y una *cota inferior máxima* glb (por sus siglas en inglés *greatest lower bond*) para cada subconjunto X de L.

En el retículo $2^{\mathsf{B}_\mathsf{P}}$ bajo la relación de inclusión de conjuntos, la mínima cota superior de cualquier conjunto de interpretaciones de Herbrand es la interpretación de Herbrand que se obtiene por la unión de todas las interpretaciones de Herbrand en el conjunto, mientras que la máxima cota inferior se obtiene por la intersección de todas las interpretaciones de Herbrand del conjunto.

Definición. Sea L un retículo completo y $\mathsf{T}{:}L \to L$ un mapeo. Decimos que T es *monotónico* si $\mathsf{T}(x) \subseteq \mathsf{T}(y)$ siempre que $x \subseteq y$.

Definición. Sea L un retículo completo y $X \subseteq L$. Decimos que X es *dirigido* si cada subconjunto finito de X tiene una cota superior en X.

Proposición. Sea X un subconjunto dirigido de $2^{\mathsf{B}_\mathsf{P}}$. Entonces, $\{\mathsf{<A_1/>},...,\mathsf{<A_n/>}\} \subseteq \mathsf{I}$ si y solo si $\{\mathsf{<A_1/>},...,\mathsf{<A_n/>}\} \subseteq \text{lub}(X)$ para alguna I en X.

Definición. Sea L un retículo completo y $\mathsf{T}{:}L \to L$ un mapeo. Decimos que T es *continuo* si $\mathsf{T}(\text{lub}(X)) = \text{lub}(\mathsf{T}(X))$ para cada subconjunto dirigido X de L.

Proposición. Sea P un programa definido. Entonces el mapeo T_P es *continuo*.

Demostración. Sea X un subconjunto dirigido de 2^{B_P}. Entonces, las siguientes afirmaciones son mutuamente equivalentes:

$<A/> \in T_P(\text{lub}(X))$

<A><A1/>...<An/></A> es la instancia base de una cláusula en P y $\{$<A1/>,...,<An/>$\} \subseteq \text{lub}(X)$

<A><A1/>...<An/></A> es la instancia base de una cláusula en P y $\{$<A1/>,...,<An/>$\} \subseteq$ I para alguna I en X

<A/> $\in T_P(I)$ para alguna I en X

$<A/> \in \text{lub}(T_P(X))$

∇

Proposición. Sea P un programa definido e I una interpretación de Herbrand de P. Entonces I es un modelo de Herbrand para P si y sólo si $T_P(I) \subseteq I$.

Demostración. Sea <A/> $\in T_P(I)$. Entonces, existe una instancia base de una cláusula <A><A1/>...<An/></A> de P, con $n \geq 0$, tal que $\{$<A1/>,...,<An/>$\} \subseteq$ I. Puesto que <A/> es consecuencia lógica de $\{$<A1/>,...,<An/>$\}$, entonces I es un modelo para <A/>, por lo que <A/> $\in$ I. Por lo tanto, $T_P(I) \subseteq I$. ∇

Definición. Sea L una retícula completa y $T{:}L \to L$ un mapeo. Decimos que $a \in L$ es el *punto fijo mínimo* de T, escrito lfp(T), si $T(a) = a$ y para todo punto fijo b de T, se cumple que $a \leq b$.

Proposición. Sea L una retícula completa y $T{:}L \to L$ un mapeo monotónico. Entonces, T tiene un punto fijo mínimo tal que $\text{lfp}(T) = \text{glb}\{x : T(x) = x\} = \text{glb}\{x : T(x) \leq x\}$.

Proposición. Sea L una retícula completa y $T{:}L \to L$ un mapeo monotónico. Supóngase que $a \in L$ y $a \leq T(a)$. Entonces, existe un punto fijo b de T tal que $a \leq b$.

Definición. Sea L una retícula completa y $T{:}L \to L$ un mapeo monotónico. Las *potencias ordinales* de T se definen de la siguiente forma:

$T \uparrow 0 = \varnothing$

$T \uparrow \alpha = T\,(T \uparrow (\alpha\text{-}1))$ si α es un ordinal sucesor

$T \uparrow \alpha = \text{lub}\{\, T \uparrow \beta : \beta < \alpha\}$ si α es un ordinal límite

Proposición. Sea L una retícula completa y $T{:}L \to L$ un mapeo monotónico. Entonces, para cualquier ordinal $T \uparrow \alpha \leq \text{lfp}(T)$.

Proposición. Sea L una retícula completa y $T{:}L \to L$ un mapeo continuo. Entonces, $\text{lfp}(T_P) = T_P \uparrow \omega$.

Teorema (Caracterización de punto fijo del modelo mínimo de Herbrand). Sea P una programa definido. Entonces $M_P = \text{lfp}(T_P) = T_P \uparrow \omega$.

Demostración.

$M_P = \cap\, \{M : \text{para todo modelo } M \text{ de Herbrand para } P\}$

$= \text{glb}\{M : M \text{ es un modelo de Herbrand para } P\}$

$= \text{glb}\{M : T_P(M) \subseteq M\}$

$= \text{lfp}(T_P)$

$= T_P \uparrow \omega$

∇

Respuesta calculada

Definición. Sea P un programa definido y G una meta. Una *respuesta* para $P \cup \{G\}$ es una sustitución para las variables de G.

Definición. Sea P un programa definido y G una meta definida <query> <A1.../>...<Ak.../></query> y θ una respuesta para $P \cup \{G\}$. La *respuesta correcta* θ para es aquella para la que (<query><A1.../>...<Ak.../></query>) θ es consecuencia lógica de P.

Teorema. Sea P una programa y G la meta <query><A1.../>...<Ak.../></query>. Suponemos que θ es una respuesta para

P ∪ {G} tal que (<A1.../>...<Ak.../>) θ es una meta base. Entonces las siguientes afirmaciones son equivalentes:

a) θ es correcta.

b) (<A1.../>...<Ak.../>) θ es verdadera con respecto a cada modelo de Herbrand para P.

c) (<A1.../>...<Ak.../>) θ es verdadera con respecto al modelo mínimo de Herbrand para P.

Demostración.

1. Si θ es correcta, (<query><A1.../>...<Ak.../></query>) θ es consecuencia lógica de P. Entonces tiene un modelo y por lo tanto tiene un modelo de Herbrand.
2. Si (<A1.../>...<Ak.../>) θ es verdadera con respecto a cada modelo de Herbrand para P, entonces es verdadera con respecto a la intersección de todos ellos. Entonces es verdadera con respecto al modelo mínimo de Herbrand para P.
3. (<A1.../>...<Ak.../>) θ es verdadera con respecto al modelo mínimo de Herbrand para P. Entonces, (<A1.../>...<Ak.../>) θ es verdadera con respecto a todos los modelos de Herbrand. De aquí que ¬ (<A1.../>...<Ak.../>) θ es falsa con respecto a todos los modelos de Herbrand. Luego, P ∪ {¬ (<A1.../>...<Ak.../>) θ} no tiene modelos de Herbrand. Entonces, P ∪ {¬ (<A1.../>...<Ak.../>) θ} no tiene modelos por lo que es insatisfacible. Por lo tanto, (<A1.../>...<Ak.../>) θ es consecuencia lógica de P, así que θ es la respuesta correcta.

∇

4.4.2 Resolución SLD

Existen varios procedimientos de refutación basados en la regla de inferencia de resolución que son refinamientos del procedimiento original de Robinson. El procedimiento de refutación de interés en este trabajo fue descrito por primera vez por Kowalski y fue llamado resolución SLD. La resolución SLD establece a la

resolución SLD para procedimientos. Resolución SLD es un descendiente directo del procedimiento de eliminación de modelos de Loveland.

Definición. Sea G la meta <query><A_1.../>...<A_m.../>...<A_k.../></query> y sea C la cláusula <A ...><B_1…/>...<B_q.../><A/>. Entonces, la meta G' se deriva de G y C usando el unificador más general (mgu) θ si se dan las siguientes condiciones:

a) <A_m.../> es un átomo, llamado el *átomo seleccionado*, en G.
b) θ es el mgu de <A_m.../> y <A.../>.
c) G' es la meta (<query> <A_1.../>... <A_{m-1}.../> <B_1.../>... <B_q.../> <A_{m+1}.../>... <A_k.../> </query>) θ.

En la terminología del cálculo de resolución, a G' se le llama el *resolvente* de G y C.

Definición. Sea P un programa definido y G una meta definida. Una *derivación SLD* de $P \cup \{G\}$ consiste de una secuencia $G_0 = G$, G_1,... finita o infinita de metas, una secuencia C_1,C_2,... de variantes de cláusulas de P y una secuencia θ_1, θ_2,... de mgu's tales que cada meta G_{i+1} se deriva de G_i y C_{i+1} usando θ_{i+1}.

Cada C_i es una variante apropiada de la correspondiente cláusula de programa de modo que C_i no tiene variables que aparezcan en la derivación de G_{i-1}. Para evitar nombres repetidos, las variables en G pueden usar 0 como subíndice mientras que las variables en C_i pueden usar i como subíndice. A este proceso de renombramiento de variables se le llama *estandarización separada de variables.*

Definición. Una *refutación SLD* de $P \cup \{G\}$ es una derivación SLD finita de $P \cup \{G\}$ que tiene una cláusula vacía como la última meta en la derivación. Si G_n = empty, se dice que la refutación tiene *longitud n.*

Definición. Una *refutación SLD no restringida* es una refutación SLD, excepto que se excluye el requerimiento de que la sustitución θ_i sea el unificador más general. Las substituciones sólo requieren ser unificadores.

En lo que sigue, se usará simplemente derivación y refutación por derivación SLD y refutación SLD. El primer resultado importante sobre el procedimiento de resolución será mostrar su solidez.

4.4.3 Solidez de la resolución SLD

Una derivación SLD puede ser finita o infinita. La Fig. 5 muestra una derivación para el problema de la relación de parentesco. Una derivación finita puede ser exitosa o fallida. Una *derivación exitosa* es aquella que termina con la cláusula vacía empty. En otras palabras, una derivación exitosa es una refutación. Una *derivación fallida* es aquella que termina con un a meta no vacía con la propiedad de que el átomo seleccionado en esta meta no se unifica con la cabecera de cualquier cláusula del programa.

Definición. Sea P un programa definido. El *conjunto exitoso* de P es el conjunto de todos átomos <A.../>$\in B_P$ tales que P $\cup$ {<query> <A.../> </query>} tienen una refutación SLD.

Un conjunto exitoso es la contraparte de la interpretación de procedimiento del modelo mínimo de Herbrand. Mostraremos que el conjunto exitoso es igual al modelo mínino de Herbrand del programa. En forma similar, también tenemos la contraparte operacional de una respuesta correcta.

Definición. Sea P un programa definido y G una meta definida. Una *respuesta computada* θ para P $\cup$ {G} es la sustitución que se obtiene al restringir la composición $\theta_1 \cdots \theta_n$ a las variables de G, donde $\theta_1,...,\theta_n$ es la secuencia de mgu's usada en la refutación SLD de P $\cup$ {G}.

La solidez de la resolución SLD consiste en probar que las respuestas computadas son correctas.

Teorema (Solidez de la resolución SLD). Sea P un programa definido y G una meta definida. Entonces, toda respuesta computada para P $\cup$ {G} es una respuesta correcta para P $\cup$ {G}.

Demostración. Sea G la meta <query><A_1.../>...<A_m.../>...<A_k.../></query> y θ_1,...,θ_n es la secuencia de mgu's usada en la refutación SLD de P $\cup$ {G}. Mostraremos, por inducción en la longitud de la refutación, que (<query><A_1.../>...<A_m.../>...<A_k.../></query>) θ_1,...,θ_n es consecuencia lógica de P.

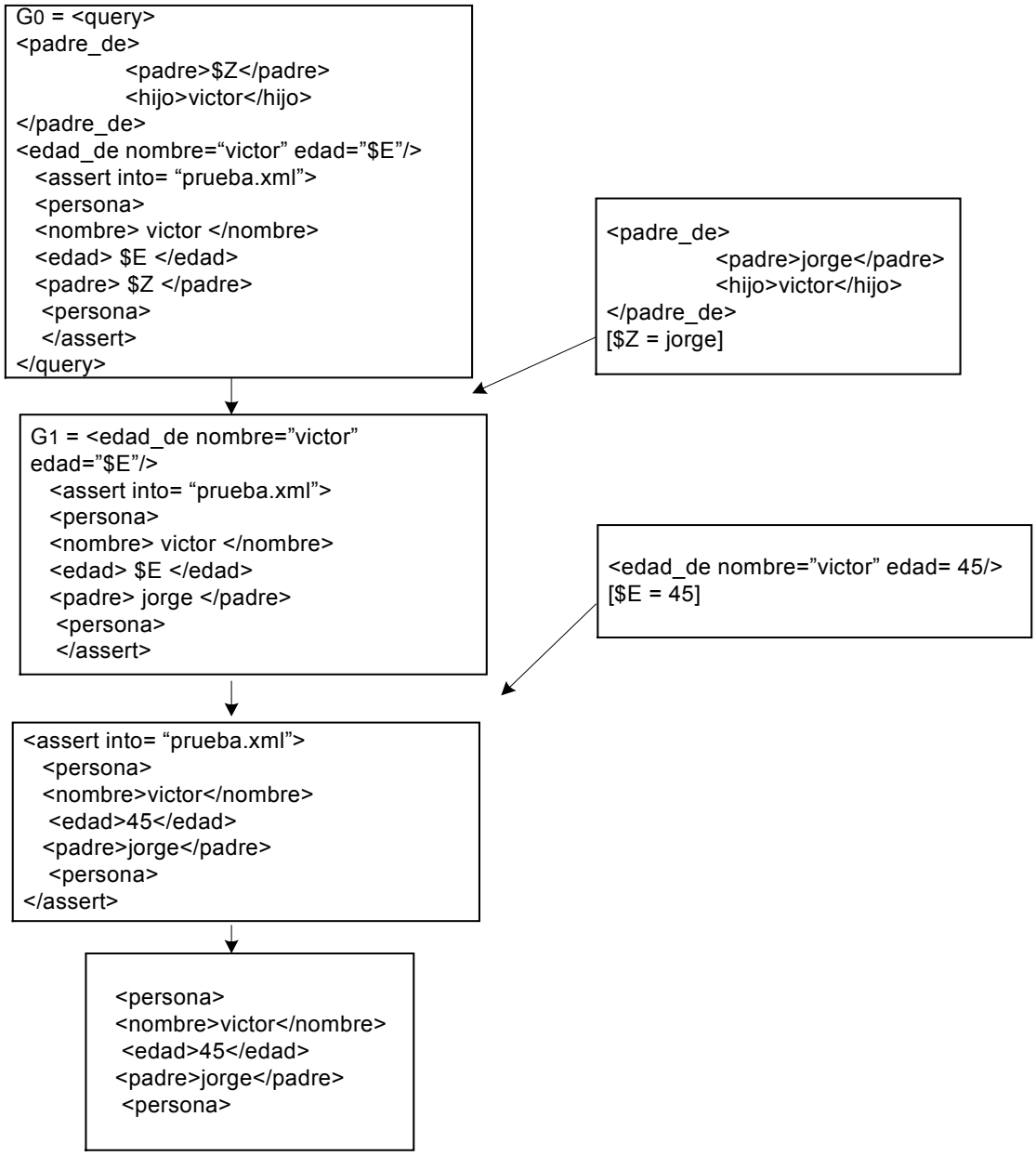

Fig. 5 Una derivación SLD

Paso base. En este caso, se tiene que G es la meta <query><A_1.../></query> y que el programa debe incluir una cláusula unitaria <A.../> tal que <A_1.../> θ = <A.../> θ. Puesto que <A.../> θ es instancia de una cláusula unitaria de P, es <A_1.../> θ consecuencia lógica de P. La respuesta computada θ es entonces la respuesta correcta.

Paso de inducción. Supongamos que el resultado se cumple para una refutación de longitud n-1. Sea $\theta_1,...,\theta_n$ es la secuencia de mgu's usada en la refutación SLD de P $\cup$ {G}. Sea <A...><B_1.../>...<B_q.../></A> ($q \geq 0$) la primer cláusula de entrada y <A_m.../> el átomo seleccionado de G. Por hipótesis de inducción, la meta (<query> <A_1.../>... <A_{m-1}.../> <B_1.../>... <B_q.../> <A_{m+1}.../>... <A_k.../> </query>) $\theta_1,...,\theta_n$ es consecuencia lógica de P. Por lo tanto, la submeta (<query> <B_1.../>... <B_q.../> </query>) $\theta_1,...,\theta_n$ también es consecuencia lógica de P. Entonces, el consecuente <A.../> $\theta_1,...,\theta_n$ de la cláusula de entrada es consecuencia lógica de P y también lo es <A_m.../> $\theta_1,...,\theta_n$. Usando este razonamiento para los restantes átomos de la consulta, concluimos que (<query><A_1.../>...<A_m.../>...<A_k.../></query>) $\theta_1,...,\theta_n$ es consecuencia lógica de P y que la respuesta computada $\theta_1,...,\theta_n$ es correcta. ∇

Como consecuencia del teorema de solidez de la resolución SLD se puede ver que, dados un programa definido P y una meta definida G, si existe una resolución SLD de P $\cup$ {G}, entonces P $\cup$ {G} es insatisfacible ya que G es consecuencia lógica de P.

Otro resultado relacionado es que el conjunto exitoso de un programa definido está incluido en el modelo mínimo de Herbrand. La razón es que el conjunto exitoso está formado por todos los átomos que son consecuencia lógica de P y por lo tanto están en el modelo mínimo de Herbrand.

4.4.4 Completitud de resolución SLD

En esta sección se habla de la resolución-SLD, iniciamos con dos lemas muy utilizados los cuales se pueden consultar sus demostraciones en [8].

Lema (mgu). Sea P un programa definido y G una meta definida. Supongamos que P ∪ {G} tiene una refutación SLD no restringida. Entonces P ∪ {G} tiene una refutación SLD de la misma longitud tal que. Si $\theta_1,...,\theta_n$ son los unificadores de la refutación SLD no restringida y $\theta'_1,..., \theta'_n$ son los mgu's de la refutación SLD, entonces existe una sustitución γ tal que $\theta_1 \cdots \theta_n = \theta'_1 \cdots \theta'_n \gamma$.

Lema (elevación). Sea P un programa definido, G una meta definida y θ una sustitución. Supongamos que existe una refutación SLD de P ∪ {Gθ} tal que las variables en la entrada son distintas de las variables en θ y G. Entonces, existe una refutación SLD de P ∪ {G} de la misma longitud tal que, si $\theta_1,..., \theta_n$ son los mgu's de la refutación SLD de P ∪ {Gθ} y $\theta'_1,..., \theta'_n$ son los mgu's de la refutación SLD de P ∪ {G}, entonces existe una sustitución γ tal que $\theta\theta_1,..., \theta_n = \theta'_1,..., \theta'_n\gamma$.

El siguiente teorema complementa el resultado de la solidez de la resolución SLD.

Teorema. Un conjunto exitoso de un programa definido es equivalente a su modelo mínimo de Herbrand.

Demostración. Es suficiente probar que el modelo mínimo de Herbrand de P está contenido en el conjunto exitoso de P. Supongamos que <A.../> está en el modelo mínimo de Herbrand de P. Puesto que el modelo mínimo de Herbrand es igual al punto fijo mínimo de P, existe un $n \in \omega$ tal que <A.../> $\in T_P \uparrow n$. Queda entonces probar por inducción sobre n que <A.../> $\in T_P \uparrow n$ implica P ∪ {<query><A.../></query>} tiene una refutación por lo que <A.../> estará en el conjunto exitoso.

Paso base. Si <A.../> $\in T_P \uparrow 1$, existe una cláusula unitaria cuya instancia base es igual a <A.../>. Entonces, <A.../> es consecuencia lógica de P por lo que existe

una refutación para P ∪ {<query><A.../></query>} y, por lo tanto, <A.../> está en el conjunto exitoso de P.

Paso de inducción. Supongamos que el resultado se cumple para n-1. Sea <A.../> ∈ $T_P\uparrow n$. Puesto que $T_P\uparrow n = T_P(T_P\uparrow n\text{-}1)$, existe una cláusula <B...><B_1.../>...<B_q.../></B> en P cuya instancia base cumple <A.../> = <B.../>θ para algún θ y {<B_1.../>,...,<B_q.../>} ⊆ $T_P\uparrow n$-1. Por hipótesis de inducción, para toda $i \in \{1,...,q\}$, <B_i.../>θ pertenece entonces al conjunto exitoso de P y P ∪ {(<query><B_i.../></query>)θ} poseerá una refutación. Puesto que <B_i.../>θ es una átomo base, las refutaciones se pueden combinar en la refutación de P ∪ {(<query><B_1.../>...<B_q.../></query>)θ}. De acuerdo al lema de elevación, existe una refutación para P ∪ {<query><B_1.../>...<B_q.../></query>}. De esta forma, por el lema mgu, P ∪ {<query><A.../></query>} posee una refutación no restringida porque existe una refutación para P ∪ {(<query><B.../></query>)θ}. De aquí que <A.../> está en el conjunto exitoso de P. ∇

Teorema. Sea P un programa definido y G una meta definida. Si P ∪ {G} es insatisfacible, entonces existe una refutación SLD para P ∪ {G}.

Demostración. Sea G la meta <query><A_1.../>...<A_k.../></query>. Puesto que P ∪ {G} es insatisfacible, G es falso con respecto a M_P. Por lo tanto, la negación Gθ de una instancia base de G, la conjunción (<A_1.../>...<A_k.../>)θ, es verdadera con respecto a M_P. Puesto que, para cada $i \in \{1,...,k\}$, <A_i.../>θ está en el modelo mínimo de Herbrand, posee entonces una refutación para P ∪ {<A_i.../>θ}. Puesto que cada <A_i.../>θ es un átomo base, las correspondientes refutaciones se pueden combinar en la refutación de P ∪ {Gθ}. Aplicando el lema de elevación, se concluye que P ∪ {G} posee una refutación. ∇

Lema. Sea P un programa definido y <A.../> un átomo con sus variables cuantificadas universalmente. Si <A.../> es consecuencia lógica de P, entonces existe una refutación SLD de P ∪ {<query><A.../></query>} con la sustitución identidad como respuesta computada.

Teorema (completitud de la resolución SLD). Sea P un programa definido y G una meta definida. Para cada respuesta correcta θ para P ∪ {G}, existe una respuesta computada σ para P ∪ {G} y una sustitución γ tal que θ y σγ tienen el mismo efecto en todas las variables de G.

Demostración. Sea G la meta <query><A1.../>...<Ak.../></query>. Puesto que θ es correcta, (<A1.../>...<Ak.../>)θ es consecuencia lógica de P. Entonces existe una refutación para P ∪ {<query><Ai.../></query>} tal que la respuesta computada es la identidad, para toda $i \in \{1,...,k\}$. Podemos combinar esas refutaciones en la refutación de P ∪ {Gθ} de modo que la respuesta computada sea la identidad.

Supóngase que la secuencia de mgus de la refutación de P ∪ {Gθ} es $\theta_1,...,\theta_n$. Entonces, $G\theta\theta_1,...,\theta_n = G\theta$. Por el lema de elevación, existe una refutación de P ∪ {G} con mgu's $\theta'_1,...,\theta'_n$ tales que $\theta\theta_1,...,\theta_n = \theta'_1,...,\theta'_n\gamma$, para alguna substitución γ. Sea $\theta_1,...,\theta_n$ denotado por σ una substitución restringida a las variables de G. Entonces, θ y σγ tienen el mismo efecto sobre todas las variables en G. ∇

4.4.5 Independencia de la regla de selección

En esta sección se introduce el concepto de regla de selección, la cual se usa en la selección de los átomos en la derivación SLD. Los resultados demuestran que para cualquier elección de la regla de computación, si P ∪ {G} es insatisfacible, siempre es posible encontrar una refutación usando la regla de selección. A éste resultado se le llama independencia de la regla de selección.

Definición. Una *regla de selección* es una función de un conjunto de metas a un conjunto de átomos tales que el valor de la función para una meta es un átomo, llamado el *átomo seleccionado* en la meta.

Definición. Sea P un programa definido, G una meta definida y R una regla de selección. Una *derivación SLD de* P ∪ {G} *vía* R es una derivación SLD de P ∪ {G} en la cual la regla de selección R se usa para seleccionar átomos.

Definición. Sea P un programa definido, G una meta definida y R una regla de selección. Una *refutación SLD de* P ∪ {G} *vía* R es una refutación SLD de P ∪ {G} en la cual la regla de selección R es usada para seleccionar átomos.

Definición. Sea P un programa definido, G una meta definida y R una regla de selección. Una *respuesta computada* R *de* P ∪ {G} es una respuesta computada para P ∪ {G} la cual proviene de una refutación de refutación SLD de P ∪ {G} vía R.

De acuerdo al teorema de completitud de la resolución SLD, si P ∪ {G} es insatisfacible, entonces existe una refutación de P ∪ {G}. El siguiente teorema establece que, para cualquier regla de selección R, hay una refutación vía R. Esto significa que un sistema de programación en lógica puede usar la regla de selección más conveniente.

El siguiente lema es importante para establecer la independencia de la regla de selección. En su enunciado, C^{body} denota el cuerpo de la cláusula C. La demostración del lema puede encontrarse en [8].

Lema (Lema de conmutación). Sea P un programa definido y G una meta definida. Supóngase que P ∪ {G} posee una refutación SLD $G_0 = G$, G_1, ..., G_{q-1}, G_q, G_{q+1}, ..., G_n con las cláusulas de entrada C_1, ..., C_n y mgu´s θ_1,...,θ_n. Sea

G_{q-1} = <query><A_1.../>...<A_i.../>... <A_j.../>...</query>

G_q = <query><A_1.../>...$C^{body}{}_q$... <A_j.../>...</query>

G_{q+1} = <query><A_1.../>...$C^{body}{}_q$...$C^{body}{}_{q+1}$...</query>

Entonces existe una refutación SLD de P ∪ {G} en la cual A_j se selecciona en G_{q-1} en lugar de A_i y a su vez A_i se selecciona en G_q en lugar de A_j. Además, si σ es la respuesta computada para P ∪ {G} a partir de la refutación dada y σ' es la respuesta computada para P ∪ {G} a partir de la nueva refutación, entonces Gσ es una variante de Gσ'.

Teorema (Independencia de la regla de selección). Sea P un programa definido y G una meta definida. Supóngase que existe una refutación SLD de P ∪ {G} con respuesta computada σ. Entonces, para cualquier regla de selección R, existe una refutación SLD de P ∪ {G} vía R con respuesta R computada σ' tal que Gσ' es una variante de Gσ.

Demostración. Aplicar el lema de conmutación repetidamente.

Teorema (Completitud fuerte de la resolución SLD). Sea P un programa definido, G una meta definida y R una regla de selección. Entonces, para cada respuesta correcta para P ∪ {G}, existe una respuesta σ computada vía R para P ∪ {G} y una substitución γ tal que θ y σγ tienen el mismo efecto sobre todas las variables en G.

Demostración. El teorema es consecuencia inmediata de la completitud de la resolución SLD y de la independencia de la regla de computación.

4.4.6 Procedimiento de refutación SLD

Un espacio de búsqueda es un tipo de árbol, llamado árbol SLD. Una regla de selección puede fijarse por adelantado en un árbol SLD construido usando esta regla de selección. Esto reduce el tamaño del espacio de búsqueda.

Definición. Sea P un programa definido y G una meta. Un árbol SLD para P ∪ {G} es un árbol que satisface lo siguiente:

a) Cada nodo de el árbol es una meta (posiblemente vacía).
b) El nodo raíz es G.
c) Sea <query> <A_1.../>... <A_m.../>... <A_k.../> </query> ($k \geq 1$) un nodo en el árbol y sea <A_m.../> el átomo seleccionado. Entonces, para cada cláusula de entrada <A> <B_1.../>...<B_n.../> </A> tal que <A_m.../> y <A.../> son unificables con mgu θ, el nodo tiene un hijo (<query> <A_1.../>... <B_1.../>... <B_n.../> <A_{m+1}.../>... <A_k.../> </query>) θ
d) Los nodos que son cláusulas vacías no tienen hijos.

Cada rama del árbol SLD es una derivación de $\mathsf{P} \cup \{\mathsf{G}\}$. Las ramas correspondientes a una derivación exitosa se llaman *ramas exitosas*, las ramas correspondientes a una derivación infinita se llaman *ramas infinitas* y las ramas correspondientes a las derivaciones fallidas son llamadas *ramas fallidas*.

Definición. Sea P un programa definido, G una meta definida y R una regla de selección. El árbol SLD para $\mathsf{P} \cup \{\mathsf{G}\}$ vía R es el árbol SLD para $\mathsf{P} \cup \{\mathsf{G}\}$ en que cada átomo seleccionado son estas selecciones para R.

Teorema. Sea P un programa definido, G una meta definida y R una regla de selección. Suponemos que $\mathsf{P} \cup \{\mathsf{G}\}$ es insatisfacible. Entonces, el árbol SLD para $\mathsf{P} \cup \{\mathsf{G}\}$ vía R tiene al menos una rama exitosa.

Este teorema puede reformularse de la siguiente forma.

Teorema. Sea P un programa definido, G una meta definida y R una regla de selección. Entonces toda respuesta computada θ para $\mathsf{P} \cup \{\mathsf{G}\}$ queda descrita por el árbol SLD correspondiente para $\mathsf{P} \cup \{\mathsf{G}\}$ vía R.

Descrita por el árbol SLD significa que dada θ existe una rama exitosa tal que θ es una instancia de la respuesta computada de la refutación correspondiente a esta rama.

Teorema. Sea P un programa definido y G una meta definida. Entonces, o bien todos los árboles SLD para $\mathsf{P} \cup \{\mathsf{G}\}$ tienen infinitas ramas exitosas, o bien todos los árboles SLD para $\mathsf{P} \cup \{\mathsf{G}\}$ tienen el mismo número finito de ramas exitosas.

Demostración. Mediante el lema de conmutación se puede definir una biyección entre las ramas exitosas de cualquier par de árboles SLD. ∇

Definición. Una *regla de búsqueda* es una estrategia de árboles de búsqueda para encontrar una rama exitosa. Un *procedimiento de refutación SLD* se especifica por una regla de selección junto con una regla de búsqueda.

Los sistemas de programación lógica como aquel en el que LogCIN-XML se basa usan la regla de selección que siempre toma el átomo más izquierdo en una meta junto con la regla de búsqueda por profundidad primero. La regla de búsqueda se implementa mediante una pila de metas. El estado de la pila de metas representa la rama que se está explorando. Los detalles de la implementación están ocultos en el componente Java Café que es un motor de inferencia de Prolog el cual se describirá en el capítulo 6.

4.5 Conclusiones

En el capítulo 3 se describió informalmente el lenguaje LogCIN-XML, discutiendo algunas de sus características. A continuación discutiremos el papel de la negación en la programación lógica y su importancia. Desafortunadamente, la negación no se ha incluido formalmente en el modelo de LogCIN-XML aunque se ha incorporado la negación por fallo en el prototipo experimental que se ha desarrollado. La principal razón para no incluir una formalización de la negación se encuentra en las dificultades para elegir la forma de negación apropiada ya que como se discute a continuación existe más de una.

En sistemas de representación de conocimiento reales, tales como el sistema de reglas de negocio de IBM CommonRules, basado en el formalismo de programas de lógica extendida, hay dos tipos de negación: una *negación débil* (denotada por not) que expresa la negación de la veracidad de una proposición (como en *Ana no pertenece al grupo*) y la *negación fuerte* (denotada por neg) que expresa explícitamente la falsedad de una proposición (como en *Ana se opone a pertenecer al grupo*). Un modelo para ambas formas de negación puede ser los conjuntos con particiones (subconjuntos no vacíos mutuamente exclusivos) en donde existe una partición con al menos dos subconjuntos. El modelo deseado consiste en asociar a un conjunto de la partición otro que representa explícitamente su *oposición* (desacuerdo a pertenecer). Así por ejemplo, si A, B y C forman la partición de U y

B es la *oposición* de A, entonces si $x \in B$ (falsedad explícita por oponerse a pertenecer) implica $x \notin A$.

En general, la negación fuerte implica a la débil, pero no necesariamente el recíproco es válido. Por ejemplo, $x \notin A$ (negación de veracidad) no implica que $x \in B$ ya que es posible que $x \in C$. En consecuencia, la doble negación de la forma "neg not" se reduce a la identidad (a *Ana le desagrada no pertenecer al grupo* implica que *Ana pertenece al grupo*), mientras que la forma "not neg" no se reduce (a *Ana no le desagrada pertenecer al grupo* no significa que *Ana pertenece al grupo*). En el caso de particiones con dos elementos (que siguen el principio del *tercero excluido*), ambas formas de negación colapsan en una sola. Por ejemplo, si solamente hay dos conjuntos en la partición, entonces $x \notin A$ equivale a $x \in B$ y recíprocamente.

La forma débil de la negación corresponde a la negación como falla en Prolog y al operador EXCEPT de SQL. Puesto que las tablas SQL no fueron pensadas para representar predicados incompletos, SQL no posee un operador de negación fuerte. En muchos dominios computacionales se supone que los predicados son completos (de acuerdo a la Suposición de Mundo Cerrado), 'not' se usa más frecuentemente que 'neg'. La negación fuerte es una negación de mundo abierto, puesto que en un mundo abierto tal como la Web, la negación de veracidad (o falla) de una proposición no implica su falsedad.

En el diseño de LogCIN-XML hemos adoptado un enfoque pragmático usando la negación por fallo ya que esta se encuentra generalmente disponible en la mayoría de las implementaciones de motores de inferencia de Prolog como la de Prolog-Cafe. Dejamos como trabajo futuro la formalización de la forma de negación usada en LogCIN-XML o sus combinaciones. Por ahora la forma de negación débil (negación por fallo) es suficiente para describir y ejecutar las especificaciones de un gran número de sistemas.

Capítulo 5

Diseño del sistema

En este capítulo se presenta el diseño de la arquitectura del sistema LogCIN-XML. La arquitectura posee un diseño por capas y está basada en el paradigma cliente-servidor de Internet. El sistema LogCIN-XML provee principalmente el mecanismo de recuperación de la información el cual usa un motor de inferencia para resolver las restricciones establecidas en el documento de consulta. Una vez recuperada la información se genera a partir de ella el documento XML que se envía como respuesta a la petición del usuario.

El diseño del software es un proceso que se enfoca sobre cuatro atributos distintos del programa: la arquitectura del software, el detalle procedimental, la estructura de los datos y la caracterización de la interfaz. La arquitectura del software define las relaciones entre los principales elementos estructurales del sistema. El detalle procedimental transforma los elementos estructurales en una descripción procedimental del software. La estructura de datos establece los datos que se van a requerir para implementar el sistema [11].

5.1 Arquitectura del sistema

La arquitectura general del sistema muestra los subsistemas y módulos que lo conforman, así como la relación que existe entre ellos. De manera general, la arquitectura del sistema utiliza el paradigma cliente servidor y está formada por un servidor Web donde se localiza el *servidor LogCIN-XML* encargado de atender las solicitudes de los usuarios. Las aplicaciones escritas en LogCIN-XML se encuentran formadas por una base extensional y una intencional de documentos, para más

detalle ver la sección especificación de la base de documentos en este capítulo. Estas aplicaciones pueden ubicarse en diferentes servidores, nombrados *servidores de aplicaciones*. Por esta razón el *servidor* LogCIN-XML se comunica por medio de Internet con los servidores de aplicaciones para poder proporcionar sus servicios. Los usuarios se conectan de forma remota al servidor LogCIN-XML por medio de Internet para obtener sus servicios. En la Fig. 6 se muestra la arquitectura general del sistema LogCIN-XML.

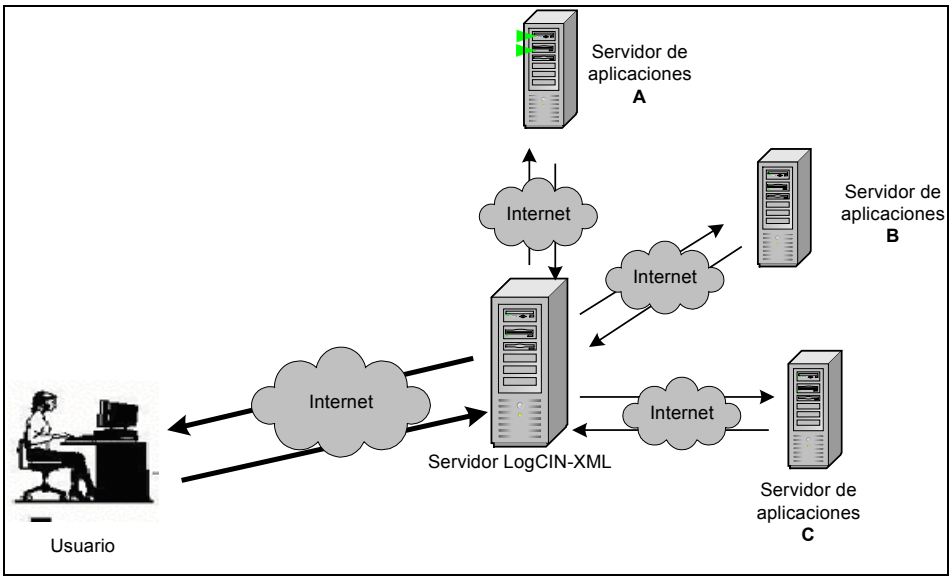

Fig. 6 Arquitectura general de LogCIN-XML

5.2 Diagrama de contexto de la arquitectura

El diagrama de contexto define todos los productores de información externos utilizados por el sistema, todos los consumidores de información externos creados por el sistema y todas las entidades que se comunican a través de la interfaz. En la Fig. 7 se muestra el diagrama de contexto de la arquitectura del sistema LogCIN-XML. En el diagrama de contexto se observa que el sistema está formado por dos subsistemas: *consultas* y *registra aplicación LogCIN-XML*. El sistema se ha divido en estos subsistemas ya que son los servicios que proporciona.

El subsistema *consultas* permite resolver consultas basándose en la información localizada en la base de documentos. El subsistema *Registra aplicación* permite incorporar o actualizar una aplicación escrita en LogCIN-XML adicionando los documento que la forman a la base de documentos. La base de documento del *servidor* LogCIN-XML está formada por bases de documentos que pertenecen a aplicaciones LogCIN-XML externas, por lo cual es necesario definir una organización a la base de documentos del servidor para que el acceso se realice de una forma eficiente.

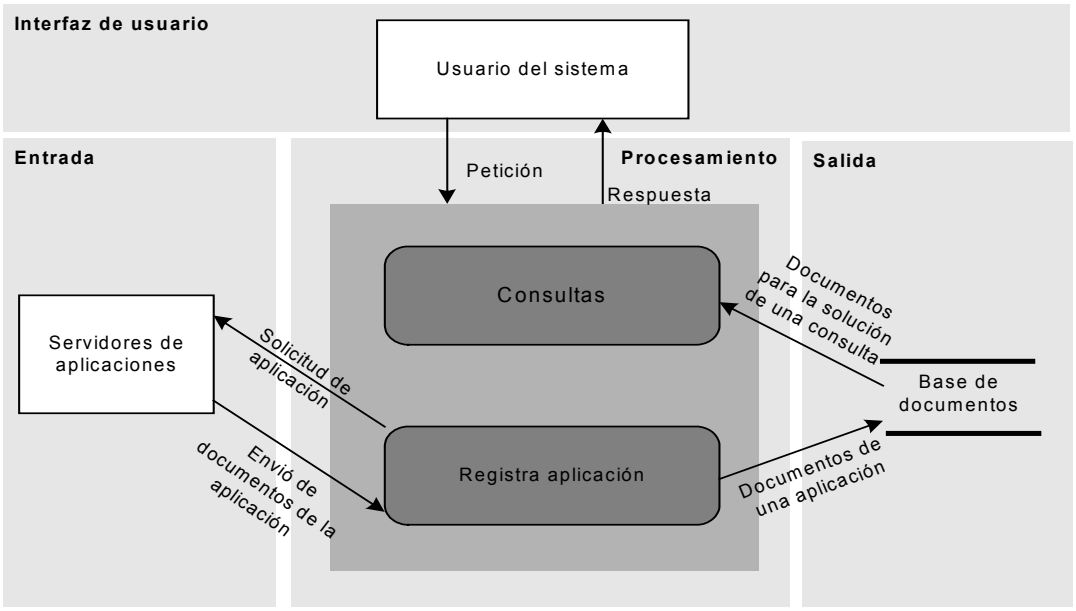

Fig. 7 Diagrama de contexto de la arquitectura del sistema LogCIN-XML

Los subsistemas principales se desglosarán a continuación dentro de los diagramas de flujo de la arquitectura correspondiente a cada uno de ellos.

5.3 Diagramas de flujos de la arquitectura del sistema

El diagrama de flujo de la arquitectura muestra los subsistemas o módulos principales y, las líneas importantes de flujo de información.

5.3.1 Consultas

El subsistema *Consultas* permite al usuario resolver una consulta definida en LogCIN-XML. El subsistema utiliza la base de documentos para poder responder la consulta y está formado por dos módulos: atiende y procesamiento LogCIN-XML.

El módulo atiende se encarga de las peticiones del usuario y envía la consulta a procesamiento LogCIN-XML para ser resuelta, una vez que obtiene la respuesta éste envía al usuario la respuesta. En la Fig. 8 se visualiza su diagrama de flujo de arquitectura correspondiente.

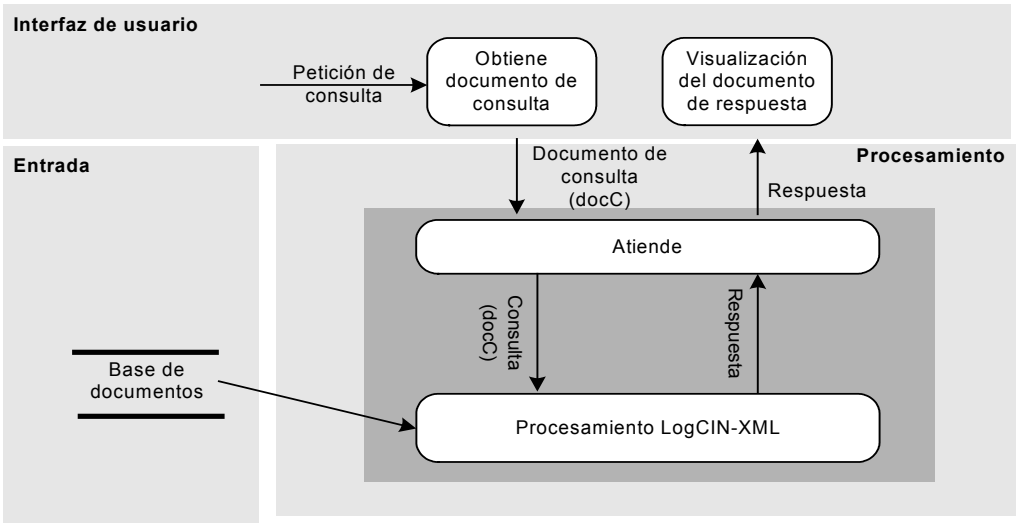

Fig. 8 Diagrama de flujo de la arquitectura del subsistema consultas

5.3.2 Registra aplicación

El subsistema *Registra aplicación* permite al usuario incorporar una aplicación. Debido a que las aplicaciones LogCIN-XML pueden ubicarse en diferentes servidores es necesario que el usuario introduzca el nombre del servidor donde radica la aplicación y el nombre de la aplicación. El usuario recibe como respuesta un reporte.

El subsistema se encuentra formado por los módulos *atiende* y *procesamiento* LogCIN-XML. El módulo *atiende* se encarga de las peticiones, además de solicitar al *servidor de aplicación* los documentos que la forman.

Después de recuperar los documentos los envía a la *máquina virtual de* LogCIN-XML para ser procesados.

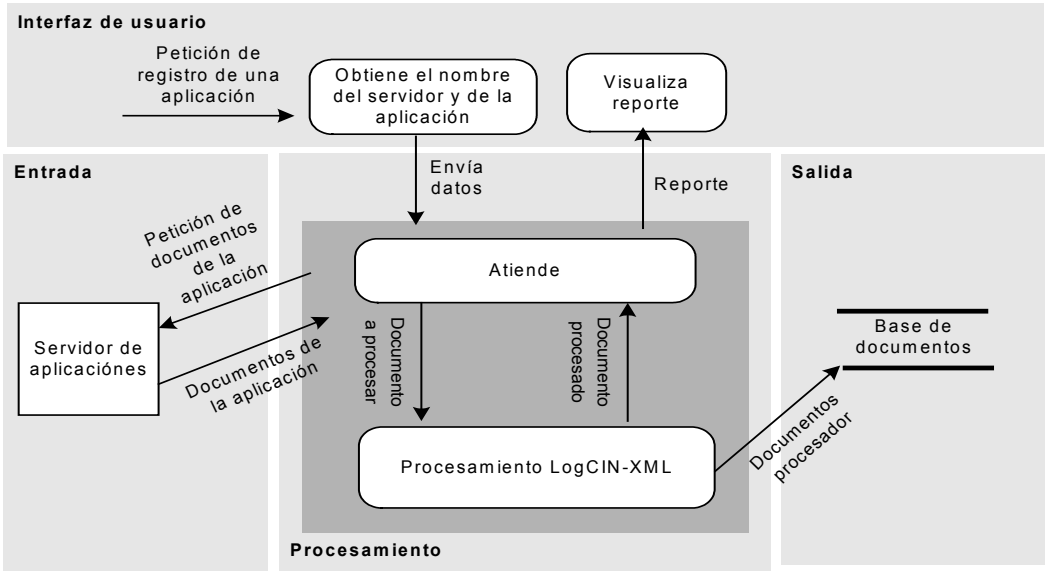

Fig. 9 Diagrama de flujo de la arquitectura del subsistema de aplicación

5.3.3 Procesamiento LogCIN-XML

El módulo procesamiento *LogCIN-XML* es uno de los módulos más importantes ya que es el encargado de traducir documentos de las bases de documentos extensionales e intencionales, así como de resolver consultas. Este módulo lo conforman tres componentes: t*raductor de* LogCIN-XML *a Prolog*, *máquina virtual de Prolog* y la *máquina virtual de Java*.

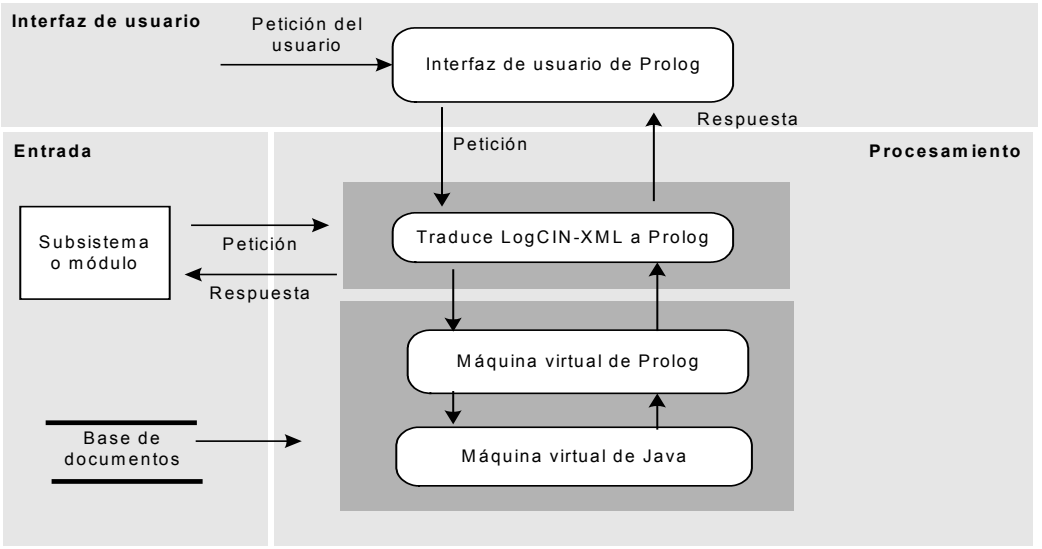

Fig. 10 Diagrama de flujo de la arquitectura del módulo procesamiento LogCIN-XML

5.4 Especificación del sistema

La especificación del software está formada por la especificación abstracta de cada subsistema.

5.4.1Consultas

Función

El subsistema *Consultas* permite al usuario realizar consultas. Este subsistema utiliza la base de documentos para poder responder la consulta y esta formada principalmente por los módulos: *atiende* y *procesamiento LogCIN-XML*.

Datos

Entrada:	Documento de consulta escrito en LogCIN-XML, Base de documentos (bases extensionales e intencionales de documentos).
Salida:	Documento XML de respuesta. La estructura de este documento es especificada dentro de la consulta.

Algoritmo

1. Recibe documento de consulta.
2. Genera la solicitud de consulta y la envía a procesamiento LogCIN-XML.
3. Si resultado_de_consulta = fail entonces

3.1 Genera documento XML <resultado> fail </resultado>.

si no

3.1 Genera documento XML a partir de resultado.

4. Fin

En el siguiente ejemplo se muestra un documento de consulta en el cual se define la estructura del documento a generarse de manera automática y las restricciones de las variables que lo conforman. Además, se muestra el documento que se genera al ser resuelto del documento de consulta.

```
<?xml version="1.0"?>
<query>
<padrede
docbase="localhost_8080/logcin/familia">
<padre>$Y</padre>
<hijo>$X</hijo>
</padrede>
<padrede>
<padre>$Z</padre>
<hijo>$Y</hijo>
</padrede>
<hijo  hijo1="#W" padre="$Y"/>
<assert
into="c://logcinserver/salida/ancestro.xml">
<ancestros>
<persona nombre = "$X"/>
<padre nombre = "$Y"/>
<abuelo nombre = "$Z"/>
#W
</ancestros>
</assert>
</query>
```

Documento de consulta

```
<?xml version="1.0" ?>
<ancestros>
<persona
nombre="roberto"></persona>
<padre nombre="fernando"></padre>
<abuelo nombre="carlos"></abuelo>
<hijo>roberto</hijo>
        </ancestros>
```

Documento de respuesta

Diagrama de flujo

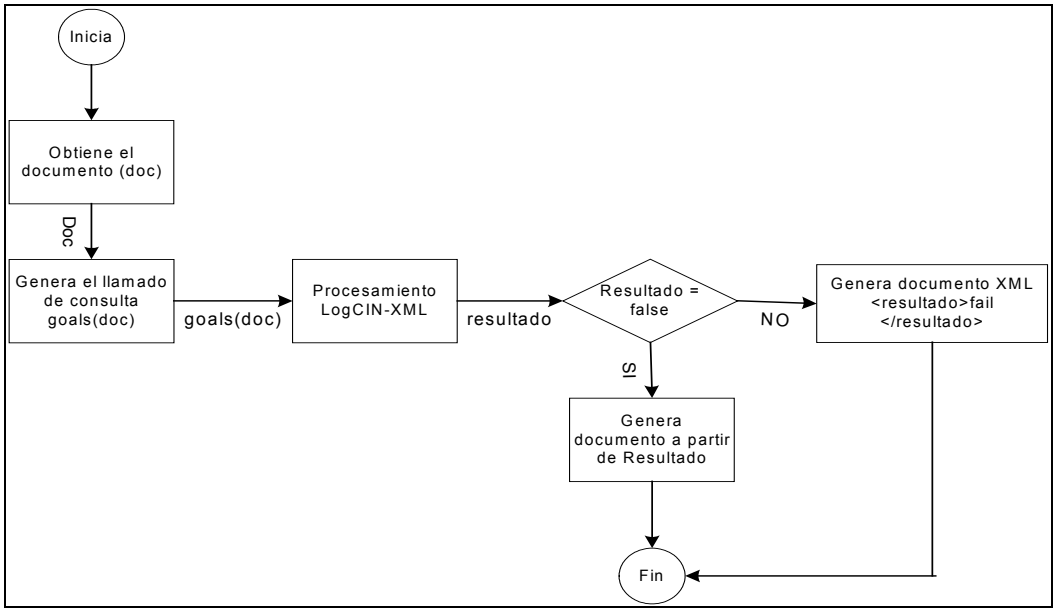

Fig. 11 Diagrama de flujo del subsistema consultas

5.4.2 Registra aplicación

Función

El subsistema *Registra aplicación* permite al usuario incorporar una aplicación al servidor LogCIN-XML. El subsistema se encuentra formado por los módulos: *atiende* y *procesamiento* LogCIN-XML. Al registrar una aplicación se crea una imagen de los documentos que forman la aplicación, la cual se agrega a la base de documentos del servidor, de esta manera los documentos originales no son modificados permitiendo que otras aplicaciones puedan trabajar con ellos. Con esto se logra tener un sistema abierto.

Datos

Entrada:	Nombre del servidor, ruta de la aplicación, documentos extensionales, documentos intencionales.
Salida:	Reporte con número de documentos encontrados, el número de documentos traducidos exitosamente y el número de documentos fallidos, almacenamiento de documentos en la base de documentos.

Algoritmo

1. Recibe nombre del servidor (nom) y ruta de la aplicación (ruta).
2. Genera solicitud al servidor de aplicación (nom+ruta).
3. Solicita documentos extensionales
4. Recibe documentos extensionales y los contabiliza (num_extensionales)
5. Para i = 1 a num_extensionales
    a. Obtiene nombre del documento extensional (i) .
    b. Genera nombre del archivo traducido.
    c. Genera llamado de traducción con el nombre del documento extensional y el nombre del archivo a traducir y envía la solicitud a procesamiento LogCIN-XML.
6. Solicita archivos intencionales
7. Recibe archivos intencionales y los contabiliza (num_intencionales)
8. Para i = 1 a num_intencionales
    a. Obtiene nombre del documento intencional(i) .
    b. Genera ruta y nombre del archivo traducido.
    c. Genera llamado de traducción con el nombre del documento intencional y el nombre del archivo a traducir y envía la solicitud a LogCIN-XML.
9. Contabiliza cuantos archivos intencionales y extensionales se encontraron, cuantos fueron traducidos exitosamente y genera el reporte a partir de esta información.
10. Fin.

5.4.3 Procesamiento LogCIN-XML

Función

Como se menciono anteriormente el módulo *procesamiento LogCIN-XML* se encuentra formado por los componentes *traductor de LogCIN-XML a Prolog*, *máquina virtual de Prolog* y *la máquina virtual de Java*. Las tareas principales de

este módulo son responder consultas basándose en la base de documentos y traducir documentos extensionales e intecionales a Prolog (imagen de los documentos).

Datos

Entrada:	Documento escrito en LogCIN-XML (doc). Bases extensionales e intecionales de documentos.
Salida:	Respuesta: Esta variable puede contener verdadero, falso o el documento generado dependiendo de la ejecución del programa.
Variables de proceso:	Body: variable que contiene la consulta en términos de Prolog. Documento: variable que contiene el documento en términos de Prolog generado al encontrar la etiqueta document. Filename: variable que contiene el nombre físico del archivo a generarse. Fileout: variable que contiene la ruta y el nombre del archivo donde se ubicará la traducción del documento.

Algoritmo

1. Recibe la petición y el documento LogCIN-XML(doc).
2. Si petición = consulta entonces
    2.1 Traduce doc en una consulta en Prolog (Body).
        2.1.1 Si <document> entonces
            2.1.1.1 Documento = genera documento.
            2.1.1.2 Respuesta = documento.
        2.1.2 Si atributo filename entonces
            2.1.2.1 Escribe Documento en filename
        Si no
            2.1.2.1Respuesta = Documento.
    2.2 Resuelve consulta en basándose en la base de documentos.
    2.3 Si falla consulta entonces
            2.3.1 Respuesta = false.
3. Si petición = traduce_extensional entonces
    3.1 Obtiene ruta y nombre del archivo traducido (Fileout).

3.2 Traduce doc a un hecho de Prolog.

3.3 Escribe el documento traducido en Fileout.

3.4 Si falla traducción entonces

3.4.1 Respuesta = false.

Si no

3.4.1 Respuesta = true.

4. Si petición = traduce_intencional

4.1 Obtiene ruta y nombre del archivo traducido (Fileout)

4.2 Traduce doc a un programa Prolog.

4.3 Escribe el documento traducido en Fileout

4.4 Si falla traducción o escritura entonces

4.4.1 Respuesta = false.

Si no

4.4.1Respuesta = true.

5. Regresa respuesta.
6. Fin.

5.5 Especificación de la Base de documentos

La base de documentos está formada por bases extensionales e intencionales de documentos. Estas bases de documentos corresponden a las bases de documentos de aplicaciones LogCIN-XML.

5.5.1 Bases extensionales de documentos

Las bases extensionales de documentos corresponden a colecciones de documentos que cumplen con el estándar XML. A continuación se visualiza un documento en la base extensional de documentos.

```
<datos_personales>
    <nombre>roberto</nombre>
    <edad>38</edad>
    <estado_civil>soltero</estado_civil>
    <ocupacion>ingeniero civil</ocupacion>
    <sexo>Masculino</sexo>
    <fecha_de_nacimiento>
      <dia>27</dia>
      <mes>08</mes>
      <ano>1965</ano>
    </fecha_de_nacimiento>
</datos_personales>
```

Documento en la base extensional

5.5.2 Bases intencionales de documentos

Las bases intencionales de documentos corresponden a colecciones de programas LogCIN-XML. Su propósito es doble: (i) construir una sub-capa inferior que describa las relaciones conceptuales que subyacen en la aplicación y (ii) construir una sub-capa superior que describa y de soporte a procedimientos comunes tales como la formulación de consultas y la generación de documentos a partir de la información que se puede inferir del contenido de las bases extensionales. A continuación se muestra un documento en la base intencional de documentos.

```
<?xml version="1.0"?>
<fecha_de_nacimientode nombre="$X" fecha="#F">
  <datos_personales>
    <nombre>$X</nombre>
    #E
    #C
    #O
    #S
    #F
  </datos_personales>
</fecha_de_nacimientode>
```

Documento en la base intencional de documentos

5.5.3 Organización de la base de documentos del servidor

Como se menciono anteriormente es necesario definir una organización a la base de documentos para que el acceso se realice de una forma eficiente. La organización establecida debe permitir una rápida localización de los documentos que pertenecen a una aplicación LogCIN-XML .

Las aplicaciones LogCIN-XML están formadas por documentos extensionales los cuales se encuentran en la carpeta bdx y los documentos intencionales que se encuentran en la carpeta bdi. Estas aplicaciones se localizan en servidores Web conectados a Internet. Por lo cual, una aplicación LogCIN-XML se puede acceder mediante su dirección URL (Uniform Resource Locator). Esta dirección nos permite establecer una organización mediante la estructura de árbol de los directorios, de esta manera, la dirección URL de una aplicación es reflejada en la estructura de directorios donde se localizan los documentos de la aplicación. Al utilizar la dirección URL de las aplicaciones evitamos las ambigüedades y la posibilidad de que dos aplicaciones se puedan representar con un mismo árbol de directorios.

En la carpeta www se localiza la base de documentos, y es aquí donde se encuentran los documentos de las aplicaciones que han sido registradas en el servidor LogCIN-XML. Las siguientes carpetas están estrechamente relacionadas con la dirección URL de la aplicación LogCIN-XML. Las direcciones URL siguen la siguiente sintaxis:

URL = servidor[“:” puerto] [ruta_absoluta]
Ejemplo: www.prueba.com/LogCIN/familia/

Esta dirección pertenece a la aplicación familia introducida en el capítulo 3. Esta aplicación se localiza en el servidor www.prueba.com con ruta LogCIN. De esta manera se asocia una carpeta con el nombre del servidor, dentro de ésta, carpetas que representa la ruta de la aplicación hasta encontrar el nombre de la aplicación, la cual contiene por finalmente las carpetas bdx y bdi. En la siguiente figura se muestra el árbol de directorios correspondiente al ejemplo anterior.

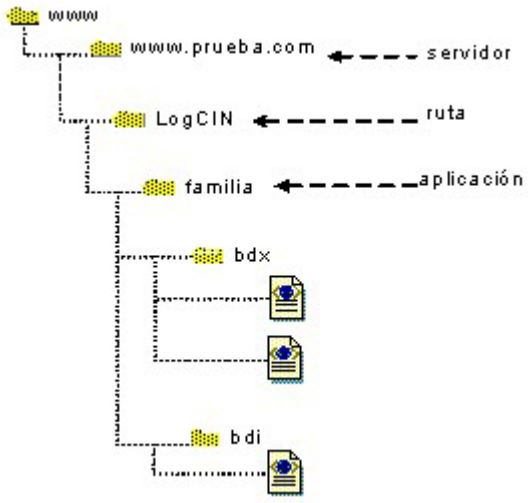

Fig. 12 Árbol de directorios correspondiente a www.prueba.com/logCIN/familia

5.6 Diseño de la interfaz

La interfaz de usuario de un sistema computacional es muy importante ya que por medio de ésta interactúa el usuario con el sistema. En la arquitectura general del sistema LogCIN-XML se visualiza que el usuario debe acceder al sistema vía Internet. Por lo cual, los servicios disponibles del servidor LogCIN-XML deben proporcionarse mediante un sitio Web. Además, al diseñar un sitio Web se tiene a los navegadores Web como programas cliente. En la Fig. 13 se muestra el mapa del sitio LogCIN-XML.

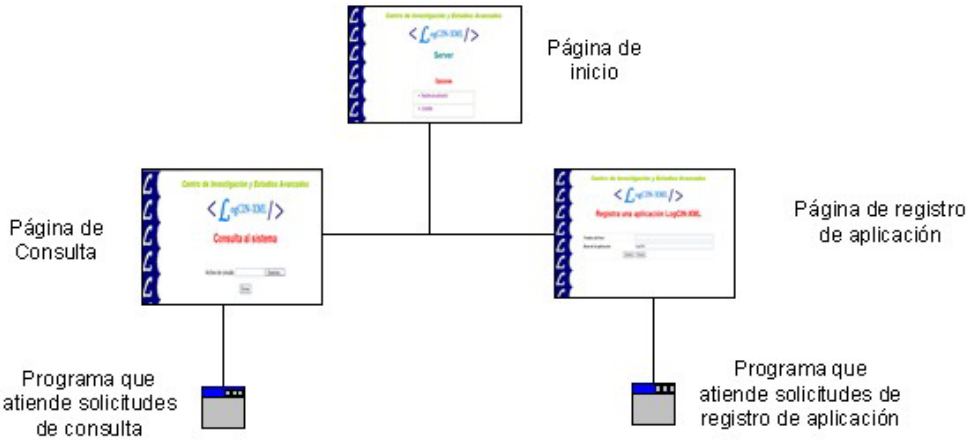

Fig. 13 Mapa del sitio LogCIN-XML

En la Fig. 14 se muestra la página de inicio. En esta página se proporcionan las opciones del *servidor LogCIN-XML*: *consulta* y *registra aplicación*.

Fig. 14 Página de inicio

En la Fig. 15 se muestra la página de consulta. En esta página se introduce la ruta y el nombre del documento de consulta a ser procesado, el usuario debe conocer el lenguaje LogCIN-XML para poder definir los documentos de consultas. Una vez que se presiona el botón *enviar* el programa que atiende las peticiones realiza el proceso de consulta y finalmente muestra el resultado obtenido en el navegador de Internet.

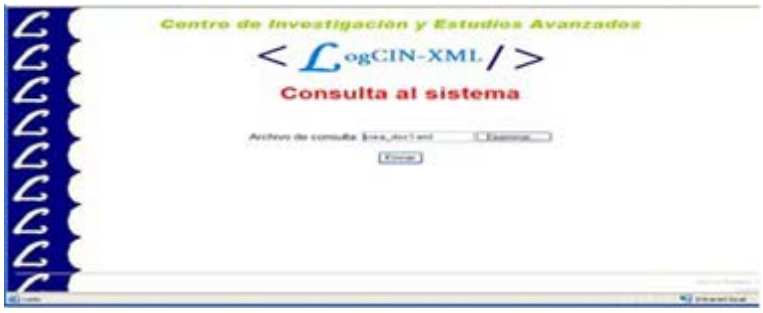

Fig. 15 Página de consulta

En la Fig. 16 se muestra la página de registro de aplicación, en esta página se introduce el nombre del servidor y la ruta incluyendo el nombre de la aplicación para poder ser localizada. Una vez que el usuario presiona el botón submit el programa que atiende las peticiones de registro se encarga de obtener la base de documento que conforman la aplicación y registrarla en el servidor LogCIN-XML.

Como respuesta en el navegador se muestra un reporte con los resultados del registro.

Fig. 16 Página de registro de aplicación

5.7 Conclusión

El diseño del sistema muestra de una forma abstracta cómo está formado el sistema. En este capítulo se cubren los cuatro atributos de un sistema computacional: y se dedica una sección del capítulo a cada uno de ellos. En la arquitectura del sistema podemos observar los subsistemas y los principales módulos que conforman el sistema. En la especificación del sistema se muestra la funcionalidad de cada subsistema y principales módulos, así como los principales datos identificados durante el diseño, esta sección pertenece al detalle procedimental. La especificación de la base de documentos permite definir los tipos de documentos que la forman y como se encuentran organizados, esta sección corresponde a la estructura de datos. Finalmente, en el diseño de la interfaz se define el tipo de interfaz que tendrá el sistema y como está conformada.

Capítulo 6

Implementación del sistema

LogCIN-XML ha sido implementado como un servidor Web de servicios de recuperación de información sobre colecciones de documentos de acuerdo al modelo formal descrito antes. Al igual que otros sistemas distribuidos en la Web, LogCIN-XML posee una arquitectura de tres partes que le permite conectarse a las aplicaciones usando protocolos estándares de comunicación de la Web. En este capítulo describiremos el motor de inferencia de LogCIN-XML, el cual es una extensión del motor de inferencia de Prolog Café, así como la integración del conjunto de clases que la componen en un servidor Apache Jakarta-Tomcat, el cual se encarga de coordinar los servicios que se prestan a las aplicaciones. La presentación comenzará describiendo cada uno de los componentes utilizados para después presentar las relaciones que guardan entre sí.

6.1 Java

Java [15] inicialmente llevó el nombre de Oak, fue desarrollado por James Gosling, en 1990. Oak fue creado para programar dispositivos electrónicos de consumo. Pero Oak fracasó. En 1995 Sun redefinió Oak como un "lenguaje de programación de Internet" portable y lo proyectó inicialmente como una forma de ejecutar programas dentro de los navegadores Web, llevando ahora el nombre de Java. La integración de Java dentro de los navegadores más populares supuso por primera vez la posibilidad de realizar uno de los principales objetivos que se habían perseguido durante años: escribir un programa en cualquier parte y que éste pudiese funcionar sin cambios en cualquier otro ordenador independientemente de su marca y del sistema operativo utilizado. El resultado fue fallido, ya que los

applets no tuvieron gran éxito. Pero Java mostró su utilidad en otras áreas particularmente en los servicios Web y la conectividad empresarial.

La máquina virtual de Java es la encargada de proporcionar la vista de un nivel de abstracción superior, donde además de la independencia de la plataforma presenta un lenguaje de programación simple, orientado a objetos, con verificación estricta de tipos de datos, múltiples hilos, con ligado dinámico y con recolección automática de basura. Una de las principales desventajas que presenta actualmente Java es su velocidad de ejecución ya que esta es muy lento. La solución que se ha propuesto es la fabricación de ordenadores que utilicen a Java como sistema operativo.

6.2 Prolog Café

Prolog Café es un sistema traductor de código Prolog a código Java. Prolog Café interactúa entre programas Java y Programas Prolog. Por el lado de Prolog, Prolog Café puede crear objetos Java como términos Prolog, invocando sus métodos, y obteniendo sus campos desde programas Prolog. Por el lado de Java, Prolog Café está formado por los paquetes jp.ac.kobe_u.cs.prolog.lang y jp.ac.kobe_u.cs.prolog.compile. Estos paquetes se encuentra comprimidos en el archivo prologcafe.jar. Prolog Café esta basado en el método de traducción jProlog y otros sistemas de traducción Prolog-Java desarrollados por Bart Demoen y Paul Tarau. El paquete jp.ac.kobe_u.cs.prolog.compiler contiene sólo la clase compiler. Esta clase contiene métodos para traducir programas Prolog a Programas Java. El paquete jp.ac.kobe_u.cs.prolog.lang implementa el sistema runtime y está formado por las clases:

DoubleTerm.- Esta clase permite definir números flotantes.
IntegerTerm.- Permite definir números enteros.
JavaObjectTerm.- Permite representar un objeto java como un término.
ListTerm.- Permite construir listas Prolog.
NumberTerm.- Es la clase abstracta de DoubleTerm e IntergerTerm.

Predicate.- Es una clase abstracta y es la súper clase de todos los predicados.
Prolog.- Esta clase implementa el motor de Prolog Café.
PrologInterface.- Esta clase implementa el modelo de flujo de control de caja de Prolog.
PrologMain.- Esta clase sólo implementa el método que inicia el sistema Prolog.
StructureTerm.- Permite definir términos compuestos.
SymbolTerm.- Permite definir átomos.
Term.- Es una clase abstracta y es la súper clase de : VariableTerm, SymbolTerm, IntegerTerm, DoubleTerm, NumberTerm, ListTerm, StructureTerm, y JavaObjectTerm.
VariableTerm .- Permite definir variables lógicas.

Además permite incorporar nuevos predicados primitivos mediante una plantilla en java.

6.3 Prolog XML

Prolog Café cuenta con la librería Prolog XML [21], la cual permite procesar documentos XML. Esta librería fue desarrollada por Tsutomu Tanizawa y trabaja con la versión 0.5.0Beta de Prolog Café y la versión 1.4 o superior de Java. Prolog XML valida un documento XML utilizando el API SAX de Java y obtiene su representación en Prolog Café. Prolog XML define un documento mediante el functor **doc** como se muestra a continuación.

doc(**xml**, [*versión*], *elementos*)
Donde:
versión corresponde a la versión del documento XML.
elementos corresponden a una lista de elementos contenidos en el documento.

Los elementos se definen por el functor **element** como se puede visualizar a continuación:

element(nombre_del_elemento, atributos_del_elemento, elementos _que_ contiene)

Donde:
elementos_que_contiene puede ser vació, una lista de elementos contenidos dentro del elemento donde se definen o el término **text**(string) que define un elemento hoja.

A continuación se muestra un documento XML y su correspondiente representación en Prolog.

<esposode> <*marido*> carlos </*marido*> <*mujer*> diana </*mujer*> </esposode>	doc(xml,[version=1.0], [element(**esposode**,[], [element(*marido*,[],[text(carlos)]), element(*mujer*,[],[text(diana)])])]).
Documento XML.	**Representación en Prolog.**

Prolog XML esta formado por los paquetes:

- jp.ac.kobe_u.cs.prolog.lang: Contiene las clases que traducen un documento XML a Prolog y viceversa, además, contiene las clases que definen los predicados encargados de leer un archivo XML y llamar a la clase que realiza la traducción.
- com.zeomtech.xmlbook : Contiene clases que se encargan de manejar los errores y clases de apoyo para obtener la representación de los documentos XML a Prolog y viceversa.
- jp.ac.kobe_u.cs.prolog.xml: Contiene las clases que definen los predicados primitivos, los cuales permiten la manipulación de los documentos XML en Prolog.

La librería Prolog XML proporciona una representación de los documentos XML, lo cual es de suma utilidad para realizar la implementación del *componente traductor LogCIN-XML*. Este proceso se esquematiza en la siguiente figura.

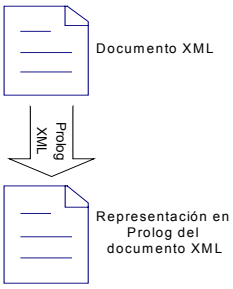

Fig. 17 Obtención de la representación en Prolog de un documento XML

Prolog XML valida los documentos XML mediante el SAX de Java y obtiene sus elementos por medio del DOM de Java para armar el término que representa al documento.

6.4 Implementación de subsistemas

A continuación se muestra la implementación de cada uno de los subsistemas que forman LogCIN-XML.

6.4.1 Consultas

El subsistema *Consultas* permite al usuario realizar consultas, está formada por los módulos: atiende y procesamiento LogCIN-XML. En está sección se encuentra la implementación del módulo atiende, la implementación del módulo procesamiento LogCIN-XML se localiza en la última sección de este capítulo. El módulo atiende se implemento en Java y extiende a la clase HttpServlet. En la siguiente figura se muestra su diagrama de clase.

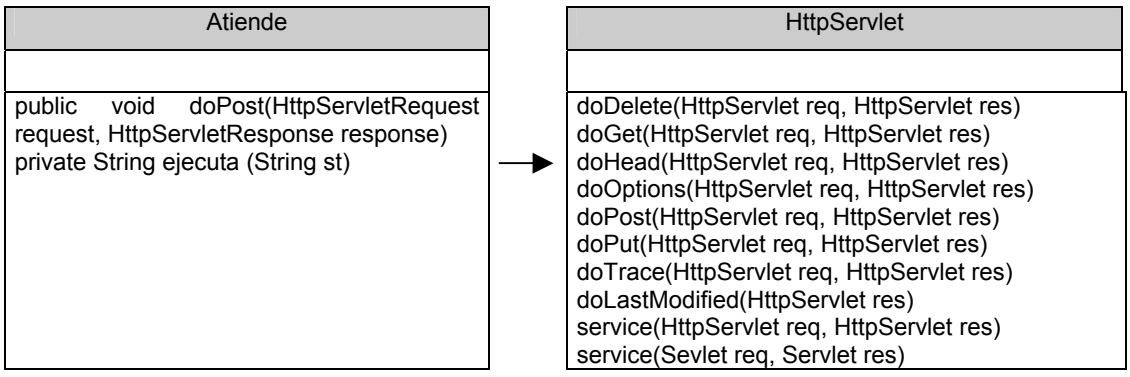

Fig. 18 Diagrama de clase de Atiende

El método doPost es el encargado de atender las solicitudes de consulta, en el objeto request se recibe la solicitud incluyendo el archivo de consulta, por esto, es necesario analizar el objeto para obtener el archivo. Una vez que se tiene el archivo se llama al método ejecuta para que resuelva la consulta. Este método regresa un String, el cual contiene la respuesta. Finalmente, ésta es mostrada al usuario.

El método ejecuta recibe en la variable st la consulta a procesar, aquí se convierte el String st a un Stream y se realiza el llamado al módulo procesamiento de *LogCIN-XML* mediante una instancia de la clase PrologInterface. Si se resuelve la consulta con éxito este método regresa el documento definido en la consulta mediante un String, de otra manera crea un documento de falla.

Para poder realizar el llamado a programas Prolog es necesario compilar los programas con el compilador de Prolog Café de esta forma se crean las clases correspondientes al programa. La desventaja que se tiene es que una vez traducido un programa Prolog a Java, éste ignora todos los programas o hechos que sean cargados de forma dinámica, trayendo como consecuencia que la respuesta obtenida sea errónea.

Debido a que en el momento de resolver una consulta es necesario realizar carga dinámica se ha decidido crear el predicado *exec/1* que nos permite llamar un programa en Prolog (llamado programa interpretado) sin la necesidad de traducir el programa a Java. El argumento de este predicado corresponde al llamado de un programa interpretado. El predicado *exec* se debe traducir a Java para poder llamarlo desde programas Java.

6.4.2 Registra aplicación

El subsistema *Registra aplicación* permite al usuario incorporar una aplicación al servidor LogCIN-XML. El módulo *registra aplicación* fue implementado en Java en la clase registra. Esta clase extiende a la clase HttpServlet y atiende las solicitudes mediante el método doPost. Este método

recibe el nombre del servidor, la ruta y nombre de la aplicación en el objeto request. Con esta información realiza la solicitud de los nombres de los documentos extensionale a la aplicación LogCIN-XML, una vez que los tiene mediante el llamado al método extensionales de la clase Logcin_read_xml se realiza la traducción de los documentos en la base extensional y los guarda en su lugar correspondiente(ver organización de la base de datos). De manera similar realiza el registro de los documentos de la base intencional. Debido a que un documento en la base intencional puede hacer referencia a otro documento, primero se analizan todos los documentos, para tener conocimiento de los documentos existentes en la base intencional. De esta manera, si un documento hace referencia a otro el sistema sabe que éste será definido posteriormente si aún no ha sido registrado. El análisis se realiza mediante el método heads de la clase Logcin_read_xml.

La solicitud del nombre de los documentos que forman una *aplicación LogCIN-XML* se realiza mediante el llamado al servlet Files, este servlet verifica la variable op, si tiene el valor uno la solicitud es de documentos en la base extensional, si el valor es dos entonces se solicitan los documento en la base intencional. A continuación se muestra como se forman las solicitudes de documentos:

Solicitud de documentos extensionales.- servidor/ruta/aplicación/Files?op=1.
Solicitud de documentos intencionale.- servidor/ruta/aplicación/Files?op=2.

A continuación se muestra su diagrama de clases del Registra, la cual implementa el modulo *registra aplicación*.

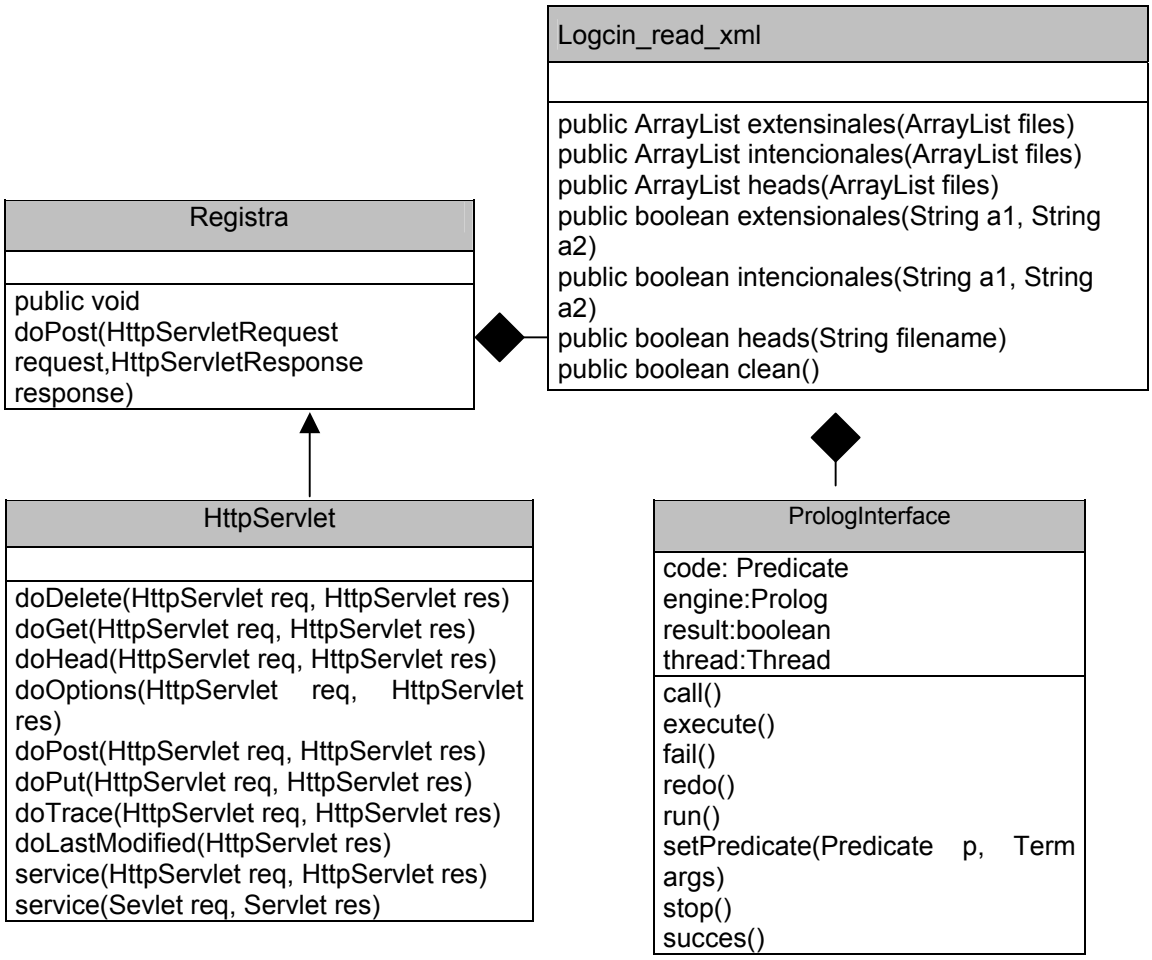

Fig. 19 Diagrama de clases de Registra

6.4.3 Procesamiento LogCIN-XML

El módulo *procesamiento de LogCIN-XML* se encuentra formado por los componentes *traductor de LogCIN-XML a Prolog*, máquina virtual de Prolog y la máquina virtual de Java. Las tareas principales de este módulo son: responder consultas basándose e la base de documentos y traducir documentos de las bases extensionales e intecionales a Prolog.

Componente máquina virtual de Java

No es necesario implementar el componente máquina virtual de Java ya que existen implementaciones de Java disponibles. La implementación que se utilizo para este sistema es Java 1.4.1_02 de Sun Microsystem.

Componente máquina virtual de Prolog

La máquina virtual de Prolog es la encargada de resolver las consultas basándose en la información localizada en la base de documentos. De manera similar al componente anterior, no es necesario implementar el componente máquina virtual de Prolog ya que existen varias implementaciones de Prolog en diferentes lenguajes. Para este sistema se utilizo Prolog Café 0.5.0Beta .

Componente traductor de LogCIN-XML

Este componente se encarga de traducir documentos escritos en el lenguaje LogCIN-XML a términos Prolog. Existen tres tipos de documentos: *de consulta*, *extensional* e *intecional*. Un documento de consulta debe ser traducido a una consulta de Prolog, un documento de la base extensional debe ser traducido a un hecho Prolog y un documento de la base intencional a una regla Prolog. El proceso de traducción se implemento en Prolog, el nombre del archivo es logcinxml.pl ubicado en la carpeta logcinserver.

Se utiliza la librería Prolog XML para obtener la representación de un documento en Prolog, pero es necesario modificarlo para que reconozca las extensiones que se incorporaron a XML. El primer concepto a integrar es el concepto de variable, al integrar este concepto también estamos integrando el concepto de término.

LogCIN-XML cuenta con variables de tipo string y término. Las variables de tipo string se identifican por el símbolo $ seguido del nombre de la variable, las variables de tipo término se identifican por el símbolo # seguido del nombre de la variable. Por está razón es necesario modificar la clase encargada de obtener la representación de un documento XML de la librería Prolog XML. La clase

encargada de realizar la traducción es XMLConverter y pertenece al paquete jp.ac.kobe_u.cs.prolog.lang. A continuación se muestra su diagrama de clase.

XMLConverter
private ignoreWhiteTextNode:Boolean =false private ignoreComment:Boolean = false private ignoreProcessingInstruction:Boolean = false private encoding:String = "" private xmlVersion:String = ""
public Document ConvertTerm(Term list) private Node ConvertTermOneNode(Term str, Document doc) private Node ConvertTermOneAttr(Term attrTerm, Document doc) public Document parse(String filename, boolean validating, boolean expandEntity) public Document parse(Reader rd, boolean validating, boolean expandEntity) public Term ConvertDOM(Document doc) public Term ConvertDOM(Document doc, boolean _ignoreWhiteTextNode, boolean _ignoreComment, boolean _ignoreProcessingInstruction) public Term ConvertDOM(Document doc, Term xmlDecl, boolean _ignoreWhiteTextNode, boolean _ignoreComment, boolean _ignoreProcessingInstruction) private Term ConvertDOMOneNode(Node node) private Term genAttrListTerm(NamedNodeMap attrmap) private Term[] genAttrUtil(String name, String value) public String getEncoding public String getXmlVersion()

Fig. 20 Clase sin modificaciones

Las modificaciones necesarias son:

- Agregar el atributo variables para almacenar las variables encontradas en el documento XML, el alcance de las variables es por documento.
- Modificar el método ConvertDOMNode. Este método convierte un nodo a su correspondiente lista en Prolog. Es necesario modificarlo para que detecte las variables localizadas en el texto de los elementos XML. La localización de variables se realiza mediante el método analiza.
- Modificar el método genAttrUtil para detectar las variables del lado de los valores de los atributos. Es importante detectar el símbolo < que nos indica que el valor del atributo corresponde a la definición de un nodo donde probablemente pueden existir nuevas variables. Si se detecta el símbolo < se

realiza una llamada al método ConvertDOM para analizar el contenido y obtener su representación en Prolog.

- Agregar el método makevariable. Este método se encarga se verificar si esta variable ya fue creada, en caso de no ser así crea una variable mediante la clase VariableTerm de Prolog Café y almacena su nombre y su referencia en el atributo variables.
- Agregar el método analiza. Este método detecta los símbolos $ y # para definir variables llamando a makevariable. En caso de no ser una variable lo considera texto, el cual es considerado un átomo en Prolog.

Una vez realizado los cambios se incorpora la clase modificada a la librería Prolog XML. En la Fig. 21 se visualiza el diagrama de clases modificado.

Después de haber realizado las modificaciones a Prolog XML en el siguiente ejemplo podemos visualizar la representación que se obtiene de un documento XML que contiene variables.

Ejemplo:

<esposode>
  <*marido*> $C </*marido*>
  #T
</esposode>
	doc(xml,[version=1.0],
	 [element(**esposode**,[],
	  [element(*marido*,[text(_1fe256b)],), _4ab8b9])]).

Documento XML con variables.	**Representación en Prolog del documento**

Traducción de extensionales

El nombre del predicado que realiza la traducción de un documento extensional es *logcin_read_xml_ext_file/2* donde el primer argumento corresponde al nombre del archivo a traducir incluida la ruta y el segundo corresponde al nombre del archivo que contendrá el documento traducido. También se tiene el predicado *logcin_read_xml_ext/2* para traducir documentos intencionales, sólo que el primer argumento corresponde a un Stream el cual contiene el documento a traducir.

Los documentos extensionales se traducen a hechos. Los hechos se crean de la siguiente manera: el elemento raíz del documento es concatenado con p_ y los establece como el nombre del predicado correspondiente a ese documento. De esta menera tenemos: p_elemento-raíz/1. La representación en Prolog del documento extensional es el argumento del predicado creado. Por cada archivo extensional existe un archivo pl. En la Fig. 22 se ejemplifica el proceso de traducción de un documento extensional.

XMLConverter
private ignoreWhiteTextNode:Boolean = false private ignoreComment:Boolean = false private ignoreProcessingInstruction:Boolean = false private encoding:String = "" private xmlVersion:String = "" private **variables**:Hastable
public Document ConvertTerm(Term list) private Node ConvertTermOneNode(Term str, Document doc) private Node ConvertTermOneAttr(Term attrTerm, Document doc) public Document parse(String filename, boolean validating, boolean expandEntity public Document parse(Reader rd, boolean validating, boolean expandEntity) public Term ConvertDOM(Document doc) public Term ConvertDOM(Document doc, boolean _ignoreWhiteTextNode, boolean _ignoreComment, boolean _ignoreProcessingInstruction) public Term ConvertDOM(Document doc, Term xmlDecl, boolean _ignoreWhiteTextNode, boolean _ignoreComment, boolean _ignoreProcessingInstruction) private Term **ConvertDOMOneNode**(Node node) private Term genAttrListTerm(NamedNodeMap attrmap) private Term[] **genAttrUti**l(String name, String value) prpublic String getEncoding public String getXmlVersion() private Term **makevariable** (String varname) private Term **analiza** (String texto)

Fig. 21 Diagrama de clase modificada

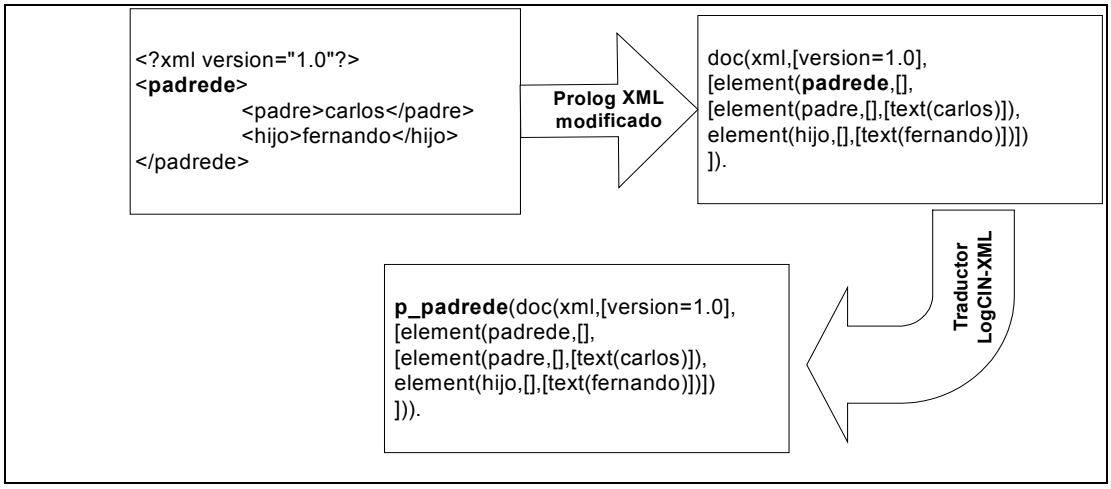

Fig. 22 Ejemplo del proceso de traducción de un documento extensional

Traducción de documentos intencionales

El nombre del predicado que realiza la traducción de un documento intencional es *logcin_read_xml_int_file/2* donde el primer argumento corresponde al nombre del archivo a traducir incluida la ruta y el segundo argumento corresponde al nombre del archivo que contendrá el documento traducido. También se tiene el predicado *logcin_read_xml_int /2* para traducir documentos intencionales, pero ahora el primer argumento corresponde a un Stream el cual contiene el documento a traducir.

Un documento intencional representa una regla por lo cual un documento debe ser traducido a una regla Prolog. Una regla está formada por dos elementos la cabecera y el cuerpo. La idea subyace en identificar el elemento raíz del documento y traducirlo en la cabecera de la regla, los argumentos del elemento raíz son ordenados por Prolog XML y se traducen en parámetros de la cabecera. Después, los elementos contenidos en el elemento raíz se traduce al cuerpo de la regla. El cuerpo de la regla puede estar formado por una o más sub-metas, por esto se realiza la traducción de cada sub-meta para formar el cuerpo. La idea se esquematiza en la siguiente figura.

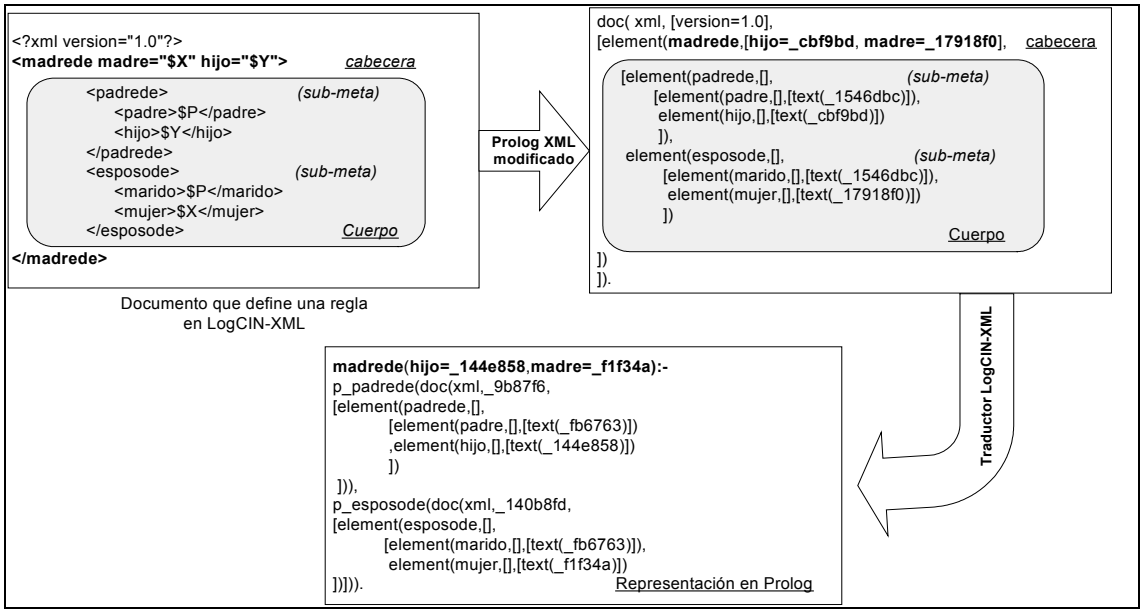

Fig. 23 Esquema del proceso de traducción de un documento intencional

Al traducir una meta primero debemos analizar se debe verificar si es la definición de un documento, el llamado de una primitiva de Prolog, el llamado a una regla, si es ninguna de estas se deduce que el llamado a un hecho. Esta diferencia se realiza ya que se traducen de diferente manera en cada caso.

Definición de un documento: En LogCIN-XML se define un documento mediante el elemento **assert**, este elemento tiene como atributo obligatorio into que corresponde al nombre que llevará el archivo del documento a generar. Una vez traducido, el documento a generar se almacena en la variable Document. Si en la definición del documento se incorporó el atributo filename la variable Document contiene el término construct(FileName, DocumentoP) donde FileName es el nombre del archivo definido y DocumentP corresponde al documento a crear en términos de Prolog. Si no se incorpora el atributo input la variable Document contiene el término construct(DocumentP).

Llamado a una primitiva de Prolog Café: Si es un llamado a una primitiva ésta es sustituida por su correspondiente llamado en Prolog. Para identificar a las primitivas se definió el término *primitive/2*, donde el primer argumento

corresponde al llamado en LogCIN-XML y el segundo contiene su equivalente en Prolog Café, por ejemplo: primitive(lessthan,<).

Llamado a una regla: la sub-meta se traduce al llamado de una regla donde el nombre del elemento corresponde al nombre de la regla a llamar y los atributos corresponden a los argumentos de la regla. Una vez traducida se verifica si esta regla ya fue definida, si no es así descarta la posibilidad de que sea una regla. Entre los atributos del llamado a una regla podemos identificar el atributo especial docbase el cual contiene la ruta de la aplicación con la debe ser resuelta esta sub-meta. Si se localiza este atributo se debe llamar al procedimiento para realizar la carga dinámica de los documentos pertenecientes a esta aplicación. El predicado *logcin_programs/2* es el encargado de realizar esta tarea, el primer argumento corresponde a la ruta docbase y el segundo regresa las instrucciones para realizar la carga dinámica. Es necesario incorporar el predicado *files/2* a Prolog Café ya que este predicado obtiene el nombre de los archivos con extensión pl existentes en una determinada ruta. El primer argumento pertenece a la ruta y el segundo argumento es una lista con los nombres de los archivos hallados. El predicado es construido en java, el nombre de la clase es PRED_files_2 y utiliza la clase filter para filtrar los archivos y obtener sólo los que tienen extensión pl. A continuación se muestra el diagrama de clases de PRED_files_2.

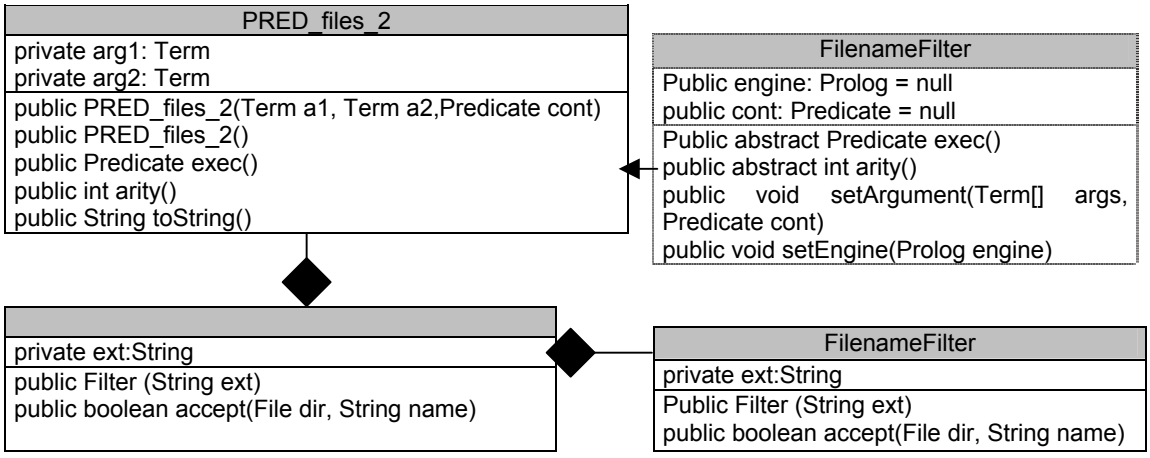

Fig. 24 Diagrama e clases de PRED_files_2

Traducción de consultas.

El nombre del predicado que realiza la traducción de una consulta es *logcin_goals_file/2* donde el primer argumento corresponde al nombre de un archivo y el segundo argumento representa el documento definido en la consulta. Esta variable se encuentra vacía si no se define un documento en la consulta. También se tiene el predicado *logcin_goals/2* para traducir una consulta. La diferencia es que el primer argumento corresponde a un Stream el cual contiene la consulta.

Una consulta esta formada por sub-metas, como anteriormente se ha implementado la traducción de sub-metas, solo se realiza el llamado a ésta para crear la consulta, una vez que se tiene, la consulta es ejecutada para ser resuelta por la máquina virtual de Prolog. Si se resuelve con éxito ya no tendremos variables en el documento definido mediante assert. Así, el documento generado será un documento XML, es decir no contendrá las extensiones a XML.

6.7 Aplicaciones LogCIN-XML

Una aplicación LogCIN-XML se implementa mediante un sitio Web y está formada por una base de documentos extensionales y una base de documentos intencionales. Estas aplicaciones deben contar con el servlet Files, que atiende las solicitudes de documentos. Su tarea principal es proporcionar los nombres de los documentos que forman la aplicación, para que posteriormente los solicitantes puedan acceder a ellos. A continuación se muestra el diagrama de clase del servlet File.

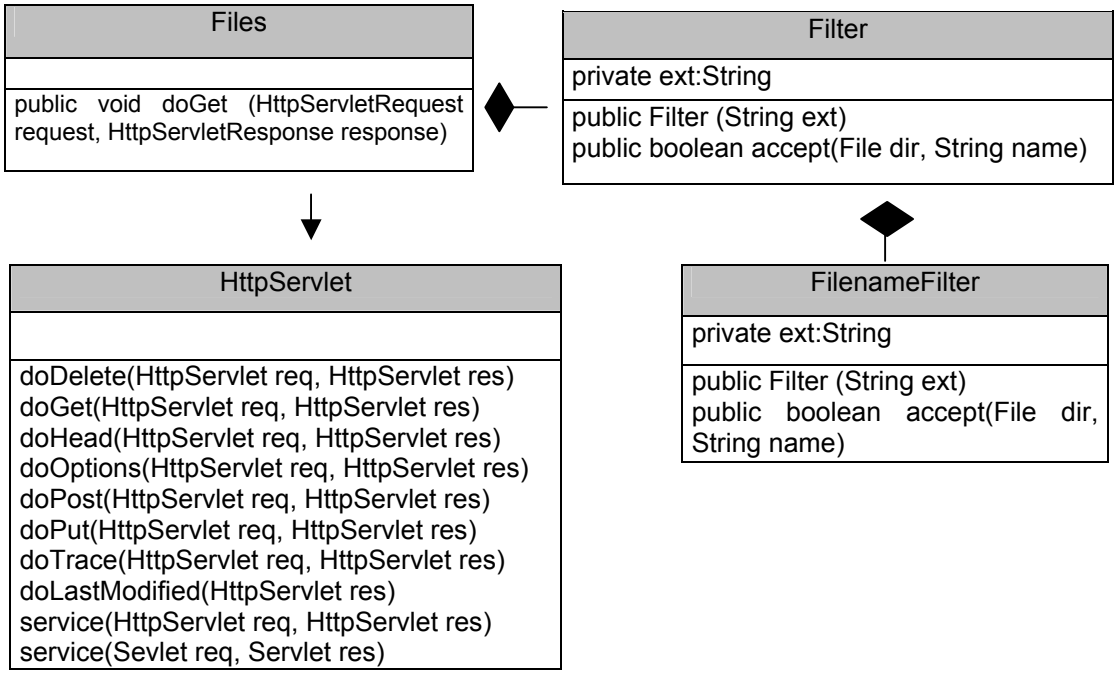

Fig. 25 Diagrama de clases del servlet File

6.8 Conclusiones

En este capítulo se pueden ver los detalles principales de la implementación del sistema LogCIN-XML. Debido a que en este capítulo ya se conoce el lenguaje en el que se encuentra implementado el sistema, se muestra información más detallada con respecto a las clases desarrolladas en Java o los programas Prolog. La finalidad de este capítulo no es mostrar el código de implementación, si no, las ideas principales y criterios que fueron tomados para su realización.

Capítulo 7

Conclusiones

En esta tesis se ha descrito el sistema LogCIN-XML que consiste de un modelo y un lenguaje de representación de la información [22,23]. El desarrollo del LogCIN-XML se ha inspirado en el modelo de coordinación HyperWorlds [24]. A partir de la información obtenida de consultas a colecciones de documentos XML, LogCIN-XML permite crear y recuperar documentos con mayor precisión que aquella obtenida por los métodos tradicionales de búsqueda. El modelo posee una semántica declarativa y operacional que permite definir y satisfacer las restricciones lógicas anotadas sobre el contenido del documento. El lenguaje consiste de un conjunto de extensiones a XML que permiten anotar las restricciones que debe cumplir el contenido del documento.

Entre las contribuciones principales de este trabajo se pueden destacar:

Las extensiones a XML para incorporar nociones fundamentales de la programación lógica como variables lógicas, procedimientos lógicos y consultas.

La definición de un modelo de representación del conocimiento adecuado para la Web Semántica.

La definición de una arquitectura abierta y distribuida que permite la integración entre aplicaciones, facilitando la interoperabilidad.

A continuación discutiremos brevemente éstas contribuciones.

LogCIN-XML extiende XML para hacer posible la programación lógica incorporándoles elementos como: variables lógicas, procedimientos lógicos y consultas. El significado de dichas construcciones sintácticas coincide con la

interpretación de una cláusula e Horn como procedimiento lógico. El paso de parámetros a estos procedimientos se hace por medio de una adaptación del algoritmo de unificación de Martelli-Montanari. El uso de unificación permite el tratamiento uniforme de varias operaciones relacionadas como la comparación de igualdad entre términos con variables, el asignamiento de términos a variables, la selección de términos y la copia de estructuras de datos complejas. Estos mecanismos aunados a los de la resolución SLD introducen un mecanismo eficaz para definir y resolver restricciones lógicas que tienen como efecto recuperar y propagar los fragmentos de documentos para generar otro. Sin duda, los mecanismos de satisfacción de restricciones tienen numerosas aplicaciones en sistemas de información y ayudarán a plantear y resolver problemas relacionados con sistemas basados en conocimientos como los sistemas e mantenimiento de la verdad LogCIN-XML es un lenguaje para la representación del conocimiento.

Aunque no ha sido diseñado como un lenguaje ontológico, LogCIN-XML posee características que facilitan la representación de servicios, procesos y modelos de negocio en forma interoperable. En LogCIN-XML se pueden representar varias restricciones preestablecidas, como aquellas de tipo y de cardinalidad así como relaciones taxonómicas como subclase, descomposición exhaustiva y descomposición disjunta. Por estar fundamentado en un lenguaje comparable al de las cláusulas de Horn, es posible describir en LogCIN.XML un conjunto restringido de fórmulas de primer orden incluyendo funciones y relaciones, restricciones de tipo y restricciones de integridad. A pesar de todas estas características, el lenguaje no posee construcciones explícitas para representar atributos de instancia, de clase y de ámbito. Sin embargo, la mayoría de ellas se pueden representar con los elementos que proporciona el lenguaje como teorías axiomáticas escritas en cláusulas de Horn.

La implementación actual ha sido instrumentada en la arquitectura cliente-servidor de Internet. El sistema es abierto, distribuido y posee un diseño por capas. El prototipo experimental reconoce las extensiones del lenguaje, da interpretación a las expresiones de acuerdo a la semántica declarativa y, como resultado, recupera

piezas de información en forma de fragmentos bien formados de documentos XML. La idea principal que subyace en la implementación del modelo propuesto en este trabajo consiste en sustituir, por una máquina de inferencia de Prolog, a la base de datos de documentos usada por los sistemas de búsqueda y recuperación de la información como Lycos, Altavista, Infoseek, WebCrawler y Google. Entre las ventajas que se pueden apreciar por adoptar este enfoque se encuentran la capacidad de derivar consecuencias lógicas tanto del contenido como de la estructura del documento Web.

Como trabajo a futuro, LogCIN-XML ha sido tomado como la base de un sistema de base de datos activa deductiva de documentos XML [25]. La reactividad de las bases de documentos XML se consigue incorporando modelos de descripción y administración de reglas evento-condición-acción. Estas reglas son una versión simplificada del lenguaje de coordinación HyperWorlds [24] el cual permite representar, ejecutar y sincronizar las acciones de sistemas de flujos de tareas y sistemas hipermedia, entre otros. Las aplicaciones que aquí se han mencionado demuestran las implicaciones que puede tener éste trabajo en numerosas áreas relacionadas. Los resultados obtenidos son prometedores. Probablemente, el trabajo requiera mayor esfuerzo de investigación sea el de incrementar la eficiencia de la implementación del actual prototipo sobre todo en el manejo de grandes bases de documentos. Esto permitiría usar LogCIN-XML como un lenguaje de descripción y recuperación de información cuyas consultas ofrecen mejores y más sofisticados resultados que aquellos obtenidos por los métodos comunes de recuperación de la información.

Bibliografía

[1] A. Gómez-Pérez, O. Corcho; "Ontology Languages for the Semantic Web INTELLIGENT SYSTEM, 54-60, 2002.

[2] V. Wuwanges, C.Anutariya, K. Akama, E. Nantajewarawat; "XML Declarative Description: A Language for the Semantic Web",IEEE INTELLIGENT SYSTEM, 54-65, 2001.

[3] J. Hendler; "Agents and the Semantic Web"; IEEE INTELLIGENT SYSTEM; 54-65; 2001.

[4] M. Chinwala, R. Malhotra, J. A. Miller; "W3C Progress towards Standards for XML database".

[5] H. Boley, S.Tabet and G.Wagner; "Design Rationale of RuleML: A Markup Language for Semantic Web Rules". In International Semantic Web Working Symposium, 2001.

[6] D. Fensel; "The Semantic Web and its language"; IEEE INTELLIGENT SYSTEM, 67-73, 2000.

[7] M.Fernandez, J. Siméon, P. Wadler; "XML Query Languages Experience and Examples"

[8] J. W. Lloyd; "Foundations of Logic Programming". Springer-Verlag, 1987.

[9] L. Sterling and E.Y. Shapiro; "The art of Prolog"; MIT Press, Cambridge MA, 1986.

[10] A. Martelli and U. Montanari; "An efficient unification algorithm"; ACM Transactions on Programming Language and Systems, 4:258-282. 1982.

[11] R. S. Pressman; "Ingeniería del software un enfoque práctico"; MacGrawHill; tercera edición, 1997.

[12] E. Rusty; "Preocessing XML with Java A guide to SAX, DOM, JDOM, JAXP and TrAX"; Addison Wesley; 2003.

[13] CH. Goldfarb, P. Prescod; "Manual de XML"; Prentice Hall; 1999.

[14] M. Morrison; "XML al descubierto"; Prentice Hall; 2000.

[15] B. Eckel; "Thinking in Java"; Prentice Hall; second edition; 2001.

[16] Prolog Café;
http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/PrologCafe061/doc/

[17] GNU; http://www.gnu.org/software/gprolog/gprolog.html

[18] JavaLog; http://sourceforge.net/projects/javalog/

[19] Jprolog; http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/

[20] NetProlog; http://netprolog.pdc.dk/

[21] Prolog XML; http://kaminari.scitec.kobe-u.ac.jp/~tanizawa/prologxml/

[22] K. Escobar, J.O. Aguirre; "Atribución de significado a documentos XML con LogCIN-XML", CIE 2003. pp. 572-581.

[23] Karina Escobar Vázquez, José Oscar Olmedo Aguirre, "Bases Deductivas de Documentos con LogCIN-XML". Memorias del 10mo Congreso Internacional de Investigación en Ciencias Computacionales CIICC'03. Oaxtepec, Morelos, México, 2003, pp. 211-218.

[24] J.O. Aguirre, K. Escobar; "El modelo de Coordinación HyperWorlds"; ANIEI; 2003.

[25] J.O. Olmedo, K. Escobar, G. Alor, G. Morales, "ADM: An Active Deductive XML Database System". Mexican International Conference on Artificial Intelligence, MICAI'2004. Springer-Verlag Heidelberg-Berlin. 2004. Aceptado para publicación.

[26] Tim Berners-Lee, James Hendler and Ora Lassila;"The Semantic Web"; Scientific American; 2001.

[27] Robinson, J. A., "A machine-oriented Logic Based on the Resolution Principle". ACM 1965, 23-41.