

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL IPN

Departamento de Ingeniería Eléctrica
Sección de Computación



**Diseño y Construcción de un Lenguaje y Modelo para Bases
Activas de Documentos XML**

Tesis que presenta
Anahi Ramírez Hernández

para obtener el grado de
Maestro en Ciencias

en la especialidad de
**Ingeniería Eléctrica,
Opción Computación**

Dirigida por el
Doctor José Oscar Olmedo Aguirre

México, D.F.

Diciembre de 2005

Resumen

Actualmente XML es el principal medio para almacenar, buscar y extraer información en un gran número de aplicaciones Web. Con el propósito de agregar mayor funcionalidad a las bases de documentos XML, la comunidad de bases de datos activas ha propuesto extenderlas para que soporten reglas ECA. También se han incorporado módulos de razonamiento deductivo que permiten derivar consecuencias lógicas a partir del contenido de la base de documentos. Los servicios Web son otra tecnología que está surgiendo para construir aplicaciones Web, en este contexto XML tiene un papel importante. Por lo anterior, los repositorios de documentos XML requieren cada vez mayor funcionalidad.

En esta tesis se hicieron dos extensiones al lenguaje ADM. Antes de este trabajo, ADM tenía características activas y deductivas para bases de documentos, sin embargo no consideraba la interacción con bases de datos relacionales. Las extensiones que se hicieron son dos: se agregó interoperabilidad entre aplicaciones que usan BD relacionales y se incorporó la tecnología de servicios Web. Para presentar la nueva funcionalidad del lenguaje se desarrolló como caso de estudio un servidor proxy de páginas Web.

La interacción de ADM con bases de datos relacionales se hizo empleando ODBC como método estándar de acceso a bases de datos. Cada ODBC se relaciona directamente con un *conector* [Agu00]. La idea principal para incorporar servicios Web al lenguaje, consistió en crear primitivas que permitan enviar y recibir fragmentos de documentos XML en el cuerpo de mensajes SOAP. La programación se hizo en Java debido a su portabilidad y las facilidades que brinda para la manipulación de documentos XML.

De las ventajas que ofrece el trabajo hecho en esta tesis se pueden considerar las siguientes: se extiende el ámbito de aplicación de las reglas ECA a la Web, se pueden invocar servicios Web de forma automática y pueden invocarse entre si. En consecuencia crece el dominio de aplicaciones de ADM, con nuestro trabajo ahora es posible automatizar workflows en Internet.

Índice general

Resumen	III
Índice General	IV
Índice de figuras	VII
Índice de tablas	x
1. Introducción	1
1.1. Motivación	1
1.2. Antecedentes	2
1.3. Planteamiento del problema	3
1.4. Solución propuesta	4
1.5. Objetivos generales y específicos	5
1.5.1. Objetivo general	5
1.5.2. Objetivos particulares	5
1.6. Metodología	6
1.7. Contribuciones	6
1.8. Estructura del documento	7
2. Bases activas de documentos XML	9
2.1. Introducción	9
2.2. XML dentro de los lenguajes de marcado	9
2.2.1. XML	9
2.3. Reglas ECA en sistemas de bases de datos	10
2.3.1. Introducción a la reglas ECA	11
2.3.2. Lenguajes de reglas de bases de datos	11
2.3.3. Vinculación entre eventos, condiciones y acciones	15
2.3.4. Ordenamiento de reglas	15
2.3.5. Semánticas de procesamiento de reglas	15
2.3.6. Arquitectura de programación	16
2.4. Trabajo Relacionado	17
2.4.1. Servicios Web y Repositorios Activos de documentos XML	17
2.4.2. Uso de servicios reactivos con reglas activas en repositorios XML	18
2.4.3. Análisis y optimización de Reglas ECA en XML	19
2.5. Resumen	21

3. Descripción Formal de ADM	23
3.1. Sintaxis	23
3.2. Modelo de Reglas Deductivas	24
3.3. Modelo de Reglas Activas	26
3.3.1. Ambito de una regla	26
3.3.2. Selección	26
3.3.3. Activación	27
3.3.4. Interacción entre Reglas Activas	28
3.3.5. Secuencialidad y Concurrencia	29
3.3.6. Comunicación Asíncrona	30
3.3.7. Comunicación Síncrona	31
3.4. Resumen	33
4. Caso de estudio: Proxy http	35
4.1. Introducción	35
4.2. Descripción del proxy http en ADM	35
4.3. Reglas ECA del proxy http	36
4.3.1. Reglas del proxy	37
4.3.2. Reglas del Servidor Interno	45
4.4. Análisis del sistema	55
4.4.1. CASO 1: Página no disponible en el proxy	55
4.4.2. CASO 2: Solicitud de página actualizada disponible en el proxy	56
4.4.3. CASO 3: Solicitud de página no actualizada disponible en el proxy	58
4.4.4. CASO 4: Página no existente	59
5. Implementación del lenguaje	63
5.1. Descripción del Administrador de Eventos	63
5.2. Conector para comunicaciones (mensajería SOAP)	63
5.2.1. Envío y recepción de objetos	64
5.2.2. Envío y recepción de objetos asociados a un identificador	64
5.2.3. Envío asíncrono	64
5.2.4. Suscripción por objetos específicos	64
5.2.5. Recepción síncrona	65
5.2.6. Aceptación de eventos compuestos	65
5.2.7. Aceptación selectiva	65
5.2.8. Aceptación transaccional	65
5.2.9. Interacción cliente-servidor	66
5.2.10. Estructuras de datos	66
5.3. Conector para colecciones (bases de datos)	72
5.3.1. Inserción, eliminación y búsqueda de objetos	73
5.3.2. Inserción, eliminación y búsqueda de objetos con identificador	73
5.3.3. Inserción y eliminación de objetos (generadora de eventos)	73
5.3.4. Inserción y eliminación silenciosa	73
5.3.5. Suscripción para recibir objetos de un tipo dado	73
5.3.6. Aceptación síncrona	73
5.3.7. Aceptación selectiva	73
5.3.8. Aceptación transaccional	74
5.3.9. Aceptación de eventos compuestos	74
5.3.10. Interacción tipo cliente-servidor	74

5.3.11. Estructuras de datos	74
5.4. Detalles de implementación	75
5.4.1. Tecnología	75
6. Conclusiones y trabajo futuro	77
Bibliografía	80
Anexos	81
A. Introducción del lenguaje	83
A.1. Reglas deductivas en ADM	83
A.1.1. Elementos XML	83
A.1.2. Términos XML	84
A.1.3. Procedimientos lógicos	85
A.2. Reglas activas en ADM	90
A.2.1. Estructura de las reglas activas	90
A.3. Resumen	91
B. Conceptos básicos y tecnologías relacionadas con XML	97
B.1. Lenguajes de marcado	97
B.1.1. SGML	97
B.1.2. HTML	97
B.1.3. XML	97
B.1.4. XHTML	98
B.2. Tecnologías relacionadas con XML	98
B.2.1. DTD	98
B.2.2. Esquemas XML	99
B.2.3. CSS	99
B.2.4. XSL	100
B.2.5. DOM	100
B.2.6. SOAP	100
B.2.7. WSDL	101

Índice de figuras

1.1. Arquitectura anterior de ADM	4
1.2. Arquitectura actual de ADM	5
3.1. Sintaxis abstracta de términos XML.	24
3.2. Substitución de variables en términos XML.	24
3.3. Unificación de términos XML.	25
3.4. Relación de inferencia SLD.	25
3.5. Correctitud de la relación de resolución SLD.	25
3.6. Selección de una regla por la ocurrencia de un evento.	27
3.7. Reglas de inferencia para la activación de reglas.	28
3.8. Reglas de inferencia para la interacción de programas.	29
3.9. Reglas de inferencia para la composición secuencial y paralela de programas.	29
3.10. Reglas de inferencia para la inserción de documentos.	31
3.11. Relación de reducción para la eliminación de documentos.	32
3.12. Reglas de inferencia para la recepción y el envío de documentos.	32
4.1. Diagrama de la regla PageReqRecByProxy	39
4.2. Diagrama de la regla DateRespRecByProxy	42
4.3. Diagrama de la regla PageRespFromIntServRecByProxy	42
4.4. Diagrama de la regla PageRespFromExtServRecByProxy	44
4.5. Diagrama de la regla ProxyAvailable	47
4.6. Diagrama de la regla DateReqRecByIntServ	47
4.7. Diagrama de la regla PageReqRecByIntServ	51
4.8. Diagrama de la regla PageCachedByIntServ	51
4.9. Diagrama de la regla PageDroppedByIntServ	54
4.10. Caso 1: Página no disponible en el proxy	57
4.11. Caso 2: Solicitud de página actualizada disponible en el proxy	58
4.12. Caso 3: Solicitud de página no actualizada disponible en el proxy	60
4.13. Caso 4: Página no existente	61
A.1. Elemento XML	84
A.2. Término XML	84
A.3. Ejemplo del meta-predicado <code>assert</code>	86
A.4. Instancia de la relación <code>padre-de</code>	86
A.5. Invocación al procedimiento de la relación <code>padre-de</code>	87
A.6. Representación de la instancia <code>padre-de</code> como procedimiento básico	87
A.7. Procedimientos generadores con variables compartidas	88
A.8. Procedimiento que define la relación <code>madre-de</code>	89

A.9. Representación de la relación madre-de como regla ECA	90
A.10.Documento de las relaciones de una familia en una base de datos intensional . . .	92
A.11.Regla activa completa	93
A.12.Contenido de la BDX extensional de las relaciones de una familia	93
A.13.Documento XML que al ser recibido activa la regla invitacion	94
A.14.Documento producido como respuesta a la activación de la regla invitacion . . .	94
A.15.Documentos de una DBX	95
B.1. Partes de un mensaje SOAP	101

Índice de cuadros

4.1. Tarjeta CRC de la clase Proxy.	37
4.2. Regla PageReqRecByProxy.	38
4.3. Regla PageReqRecByProxy en Java.	38
4.4. Regla DateRespRecByProxy.	40
4.5. Regla DateRespRecByProxy en Java.	41
4.6. Regla PageRespFromIntServRecByProxy.	43
4.7. Regla PageRespFromIntServRecByProxy en Java.	43
4.8. Regla PageRespFromExtServerRecByProxy.	44
4.9. Regla PageRespFromExtServRecByProxy en Java.	45
4.10. Regla ProxyAvailable.	45
4.11. Regla ProxyAvailable en Java.	46
4.12. Tarjeta CRC de la clase InternalServer.	46
4.13. Regla DateReqRecByIntServ.	48
4.14. Regla DateReqRecByIntServ en Java.	48
4.15. Regla PageReqRecByIntServ.	49
4.16. Regla PageReqRecByIntServ en Java.	50
4.17. Regla PageCachedByInternalServer.	52
4.18. Regla PageCachedByIntServ en Java.	53
4.19. Regla PageDroppedByInternalServer.	54
4.20. Regla PageDroppedByIntServ en Java.	55

Capítulo 1

Introducción

1.1. Motivación

Las bases de datos relacionales permiten almacenar información, mientras que los SGBD (Sistema Gestor de Bases de Datos) permiten al usuario crear y dar mantenimiento a sus bases de datos. Sin embargo los SGBD relacionales tienen algunas limitantes, por ejemplo: la dificultad que presentan para almacenar información estructurada o para interoperar entre SGBD diferentes. XML proporciona una alternativa para algunas de estas limitantes, por ello se ha incrementado su uso para almacenar e intercambiar información.

Con el propósito de agregar mayor funcionalidad a las bases de documentos XML, la comunidad de bases de datos activas ha propuesto extenderlas para que soporten reglas Evento-Condición-Acción (reglas ECA). Para tal propósito se extiende el modelo de bases de datos relacionales con la tecnología de sistemas expertos. Como resultado surgen las bases activas de documentos XML. Tales sistemas tienen importantes aplicaciones potenciales y constituyen un marco natural de integración de servicios, los cuales se ofrecen usando mecanismos separados.

Por otra parte, los servicios Web están surgiendo como un nuevo paradigma para construir aplicaciones Web. En este escenario, XML juega un papel muy importante debido a que se puede usar como fuente de datos y como una capa de interoperabilidad entre fuentes de datos convencionales. Además, usando reglas activas es posible generar o manipular el contenido de repositorios XML como una reacción a los cambios que ocurren en la información XML, lo cual ofrece una infraestructura particularmente relevante en el campo de los servicios Web.

La integración de los servicios Web con repositorios activos de documentos XML ofrece una infraestructura que proporciona grandes ventajas con respecto al uso de ambas tecnologías por separado, en seguida se mencionan algunos ejemplos:

- El ámbito de aplicación de las reglas ECA con la incorporación de servicios Web se extiende a la Web. Dado que los servicios Web se ubican en diversos URLs, cuando la acción de una regla invoca un servicio Web, es posible que dicho servicio manipule información que se encuentra sobre Internet o en máquinas diferentes a la que contiene las reglas.
- Usando reglas ECA se puede invocar servicios Web de forma automática. Los servicios Web se ejecutan mediante invocaciones que hace un cliente o alguna aplicación, usando reglas ECA es posible invocarlos automáticamente. Cuando la información en un repositorio de documentos XML se modifica y se cumple la condición de la regla ECA condición sobre la información del documento, entonces, la regla envía un mensaje en el que invoca un servicio Web.

- La respuesta de un servicio Web puede activar a otro servicio. Usando Reglas ECA es posible analizar la respuesta de los servicios Web, y en base a esa información activar reglas que invocan otros servicios.

Para integrar las reglas ECA con servicios Web, dichas reglas necesitan un mecanismo de vinculación que les permita invocar servicios Web y percibir la respuesta que estos envían. Para lograr esta vinculación se requiere incorporar dos primitivas en las reglas ECA: una primitiva que envíe las peticiones de servicios Web así como las respuestas a dichos servicios y otra primitiva que permita recibir las peticiones de los servicios Web y recibir las respuestas de dichos servicios.

Existen diversas aplicaciones que requieren el uso de bases de datos relacionales y bases de documentos además de reglas ECA. Para este tipo de aplicaciones se necesita un mecanismo dentro de las reglas ECA que les permita interactuar directamente con bases de datos relacionales empleando un método estándar de acceso a bases de datos (por ejemplo ODBC). Una vez que se tenga esta forma de comunicarse con bases de datos relacionales, se requieren además algunas primitivas que permitan consultar y modificar los datos contenidos en la base de datos a que hace referencia el ODBC.

1.2. Antecedentes

Las bases de datos (BD) permiten almacenar información dentro de los sistemas de información. Las BDs relacionales se constituyen con tres elementos: modelos de datos, lenguajes de consulta y álgebras de consulta. Los *modelos de datos* describen colecciones homogéneas de registros o tuplas agrupadas en tablas. Los *lenguajes de consulta* permiten describir las condiciones que deben cumplir los datos buscados. Las *álgebras de consulta* son un conjunto de operadores que permiten manipular y optimizar consultas.

Las Bases de Datos Deductivas (BDD) extienden las capacidades de las BDs ya que permiten hacer consultas que requieren inferencia lógica. Algunos ejemplos de BDD son Datalog, LDL, Glue-Nail y Coral entre otras. Las BDD usan la lógica de primer orden como fundamento teórico, lo que permite modelar datos, programas, consultas y restricciones de integridad. Con lo anterior se ofrece un lenguaje común para la programación de aplicaciones. En consecuencia, el usuario emplea únicamente conceptos declarativos relacionados con la aplicación.

También existen las Bases de Datos Activas (BDA) entre las cuales podemos citar Ariel, Starburst, RPL, Postgres, HiPAC y Ode. Las BDA permiten definir reglas ECA en los sistemas de bases de datos relacionales. Las reglas ECA, también conocidas como reglas de producción, son el componente que hace que los sistemas de bases de datos (o bases de documentos) tengan la característica de ser reactivos.

Dado que XML se ha convertido en el medio predominante para el intercambio de información en la Web, se introdujeron también las bases de documentos XML (BDX). Las BDX, en forma similar a las BD convencionales, también necesitan definir modelos de datos, lenguajes de consulta y álgebras de consulta. En este caso los modelos de datos (DTD, Schema, DOM, etc.) definen la estructura y las reglas para construir documentos válidos. Los lenguajes de consulta (XML-QL, Lorel, Quilt, XQuery, etc.) definen expresiones usadas para localizar y extraer datos contenidos en documentos XML. Las álgebras de consulta (álgebra Lore, YAT, álgebra IBM, álgebra ATT, etc.) definen operadores que permiten realizar evaluación eficiente, optimización y planeación de la ejecución de expresiones de consulta.

Las bases deductivas de documentos XML (BDDX) extienden el modelo de datos de XML con conceptos fundamentales de programación lógica (elementos con variables, cláusulas, consultas, etc.). Lo anterior permite hacer inferencias lógicas sobre el contenido de los documentos almacenados en una BDX. Como ejemplos de BDDX podemos mencionar XDD y LogCIN-XML.

Las bases activas de documentos XML (BADX) permiten definir reglas ECA sobre repositorios de documentos XML. Las reglas ECA proporcionan funcionalidad para modificar el contenido de la BDX o diseminar información automáticamente como respuesta a cambios percibidos en el contenido de colecciones de documentos XML.

Las bases activas deductivas de documentos XML (BADDX) buscan unificar la reactividad de las BADXs con la capacidad de derivar conclusiones lógicas de las BDDXs a partir del contenido de una colección de documentos XML.

Además de las bases de datos y las bases de documentos XML, existen otras tecnologías que permiten manipular información tales como: minería de datos, sistemas de recomendación basados en la Web y los servicios Web. Los servicios Web son aplicaciones que implementan procedimientos que están disponibles para que los clientes hagan uso de ellos. Los servicios quedan disponibles en un contenedor del lado del servidor y se ofrecen vía Web. En un escenario típico de servicios Web, una aplicación de negocios envía una petición de un servicio a un URL específico usando el protocolo SOAP sobre HTTP. El servicio recibe la petición, la procesa y devuelve una respuesta.

1.3. Planteamiento del problema

Los sistemas de BD son pasivos dado que responden únicamente a las consultas formuladas por usuarios u otras aplicaciones. Sin embargo si un sistema de BD provee mecanismos para crear y procesar reglas de producción, entonces el sistema se vuelve activo debido a que reacciona por sí mismo [HW93].

Por otro lado los sistemas de bases de datos deductivos son similares a los sistemas de bases de datos convencionales en que son pasivos ya que responden únicamente a comandos emitidos por el usuario u otras aplicaciones; estos sistemas permiten la definición de reglas al estilo de PROLOG sobre los datos y proveen una máquina de inferencia deductiva para procesar consultas recursivas usando estas reglas.

El lenguaje ADM tiene capacidades activas y deductivas para repositorios de documentos XML. Sus capacidades deductivas le permiten recuperar información a partir del contenido de una colección de documentos aunque dicha información no se encuentre de manera explícita. La capacidades activas de ADM le permiten reaccionar ante modificaciones hechas a repositorios de documentos XML.

En la figura 1.1 se muestra la arquitectura general que tenía ADM antes de las extensiones hechas en esta tesis. Se puede ver en dicha figura que ADM tenía la capacidad de trabajar únicamente con bases de documentos. Para hacer la vinculación de las DBX ubicadas en otras máquinas, ADM necesitaba una DBX interna para almacenar la información de las máquinas externas. La base de documentos interna se componía de un directorio raíz dentro del cual se insertaban directorios con información correspondiente a cada una de las bases de datos externas.

Dado que ADM no contaba con la capacidad de interoperar con aplicaciones que usan bases de datos relacionales, esto impedía la automatización de actividades que involucran tanto información de bases de datos relacionales como información de BDX. De ahí surge la necesidad de extender el lenguaje para que permita interoperabilidad de bases de documentos con las tecnologías existentes y ampliamente disponibles de bases de datos.

Como se puede ver en la figura 1.1 ADM podía atender consultas de clientes desde Internet así como acceder a información ubicada en cualquier lugar sobre Internet mediante un URL. Sin embargo como ya se mencionó, el dominio de aplicación de este lenguaje en Internet no le permitía automatizar tareas en la Web. Por ello también se propone extender el ámbito de aplicaciones de ADM a la automatización de tareas en Internet usando para ello servicios Web. El hecho de que ADM trabaje en la Web hace posible la integración de sistemas distribuidos basados

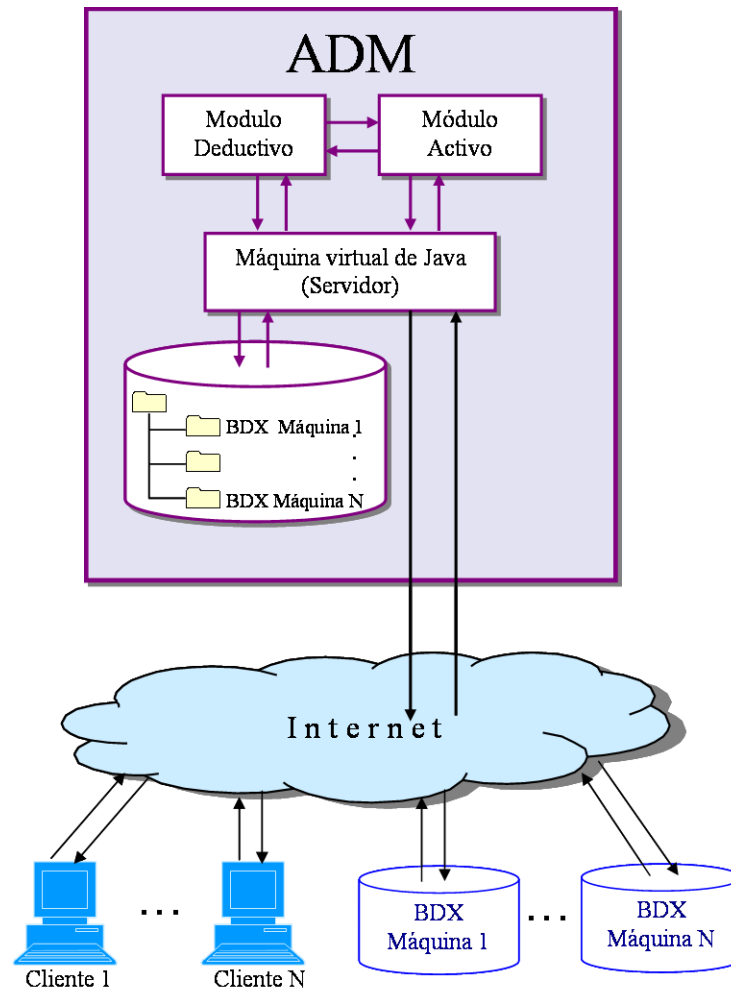


Figura 1.1: Arquitectura anterior de ADM

en documentos (por ejemplo los sistemas de workflows).

1.4. Solución propuesta

Proponemos tomar el lenguaje ADM que soporta características activas y deductivas para sistemas de bases documentos y extenderlo con las siguientes funcionalidades: interoperabilidad entre aplicaciones que usan BDs convencionales e incorporación de la tecnología de servicios Web.

Como se puede ver en la figura 1.2, ADM ya no necesita un repositorio interno de documentos para comunicarse con las BDX de máquinas externas. Las extensiones propuestas para ADM hacen posible que este lenguaje tenga la capacidad de comunicarse vía Internet con BDs convencionales.

También se puede ver en la figura 1.2, que se incorporó un mecanismo para interactuar con servicios Web, por lo que ADM puede comunicarse con Servidores de Servicios Web (SSW) vía Internet.

En este trabajo, el uso de conectores [Agu00] juega un papel muy importante ya que cada conector se vincula a una fuente de información específica, logrando así la interacción con fuentes de información de diversos tipos.

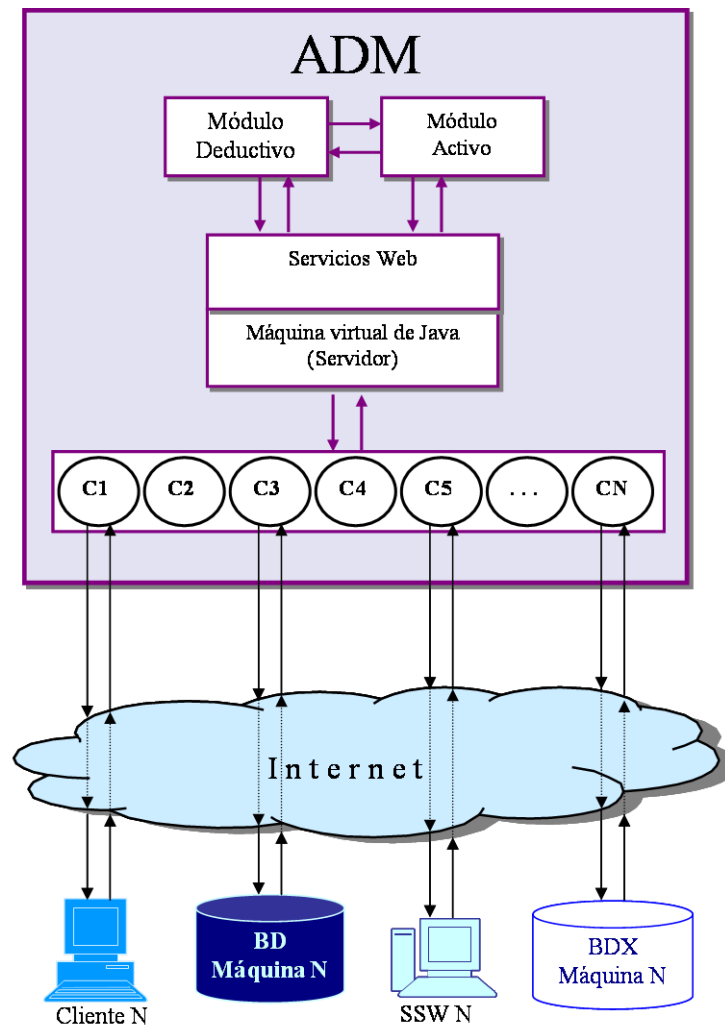


Figura 1.2: Arquitectura actual de ADM

1.5. Objetivos generales y específicos

1.5.1. Objetivo general

Extender el dominio de aplicaciones de ADM incorporando las tecnologías existentes de servicios Web y bases de datos relacionales. Dichas extensiones permitirán obtener un lenguaje y modelo apropiados para la automatización de actividades centradas en documentos que se conducen en Internet.

1.5.2. Objetivos particulares

- Definir la sintaxis de las primitivas que extienden el lenguaje
- Estudiar los enfoques de la vinculación de eventos con las condiciones y acciones en BADX con relación a su interoperabilidad, expresividad, eficiencia y métodos de análisis
- Definir la semántica operacional de las operaciones primitivas incorporadas al lenguaje
- Extender la máquina virtual que interpreta dicho lenguaje

- Desarrollar una aplicación que permita validar el diseño e implementación tanto del lenguaje como de su modelo de programación.

1.6. Metodología

El modelo de programación de ADM se puede dividir en dos módulos principales: el módulo deductivo y el módulo activo. El módulo deductivo se tomó del proyecto [Esc04], en donde se desarrolló un lenguaje para agregar capacidades de deducción a repositorios de documentos XML. Este módulo fue posteriormente revisado para eliminar varias de sus limitaciones. Para el módulo activo se diseñaron primitivas adicionales que proporcionan al sistema la capacidad de reaccionar ante eventos que suceden en los repositorios de documentos.

En relación a las extensiones del lenguaje con servicios Web, la idea esencial es enviar y recibir fragmentos de documentos XML en el cuerpo de los mensajes SOAP. Dentro del cuerpo del mensaje el nodo raíz es el nombre de la operación que proporciona un servicio Web.

Con respecto a la interacción de ADM con bases de datos relacionales, el punto principal es identificar documentos XML planos que indiquen la estructura y contenido de una base de datos relacional. Además, para hacer la vinculación del documento XML con la base de datos relacional se incorpora la noción de **conector** propuesta en [Agu00]. En este trabajo se usan los conectores de tal forma que permiten diferenciar y manejar diversos tipos de repositorios de información.

Una vez que se tuvieron los elementos necesarios se definió la semántica operacional de las extensiones que se hicieron al lenguaje para integrarlas posteriormente con la parte del lenguaje que se encarga de la reactividad y la deducción.

Finalmente se extendió la máquina virtual para que implemente la semántica operacional y se probó el funcionamiento de la máquina virtual. Para tal efecto se desarrolló una aplicación.

La aplicación que se implementó para validar la implementación de las extensiones hechas a ADM, es un servidor proxy de páginas Web conocido también como proxy-http. Un proxy-http es una computadora que almacena las páginas solicitadas por los usuarios a los servidores de Internet. De esta manera, cada vez que se solicita una página se descargará directamente del proxy-http, en lugar de hacerlo desde el servidor remoto (donde se encuentra la página original). Los proxy-http reducen el tráfico de red y la latencia para obtener datos Web debido a que los clientes pueden obtener los datos de un proxy http local en vez de pedir los datos directamente del sitio que los proporciona [IC97].

1.7. Contribuciones

1. En esta tesis se hicieron extensiones a ADM para que dicho lenguaje pueda interactuar con bases de datos convencionales. Con esta característica, es posible hacer aplicaciones que intercambien información entre bases de datos convencionales y bases de documentos XML. La interacción con las bases de datos relacionales se hace vía ODBC.

Las primitivas que permiten la interacción con bases de datos son: **insert**, **delete**, **inserted** y **deleted**. La primitiva **insert** permite insertar un registro en una bases de datos. La primitiva **delete** permite borrar un registro en una base de datos. La primitiva **inserted** reacciona ante la inserción de un registro producida por **insert**.

2. Se extendió el dominio de aplicaciones a la automatización de tareas en Internet. Ahora ADM tiene soporte para servicios Web, con esta funcionalidad, las reglas ECA pueden activarse mediante respuestas de servicios Web, o bien hacer que los servicios Web se invoquen de

forma automática. Para lograr lo anterior se incorporaron las primitivas `send` y `received`. Estas dos primitivas permiten enviar y recibir documentos vía Web.

Los puntos antes mencionados aunados al poder deductivo de ADM, dan como resultado un lenguaje que tiene capacidades activas y deductivas, que opera con bases de documentos y bases de datos, e integra la tecnología de servicios Web.

1.8. Estructura del documento

Esta tesis se encuentra estructurada de la siguiente manera: en el capítulo 2 se dan las definiciones importantes para comprender el contenido de este documento. Tales definiciones incluyen XML, las herramientas para manipular XML y definir servicios Web, las ventajas de usar XML como medio de almacenamiento y compartición de datos, conceptos básicos de las reglas ECA, y trabajo que se han implementado para dar capacidades reactivas a las BDX. En el capítulo 3 se describe el lenguaje de manera formal nuevamente tomando en cuenta la parte deductiva y la parte activa del lenguaje. Para mostrar lo anterior se desarrolló el caso de estudio de un proxy-http.

En el capítulo 4 se define el caso de estudio que se presenta en esta tesis y que se refiere a un servidor proxy de páginas Web.

En el capítulo 5 se describe a detalle la forma en que se implementó el caso de estudio y muestra los diagramas UML de secuencia que visualizan su comportamiento.

El capítulo 6 muestra las conclusiones obtenidas al realizar este proyecto y el trabajo futuro que se puede hacer partiendo del mismo.

En el apéndice A se hace una introducción al lenguaje que se desarrolló así como a la parte deductiva (ya existente antes de este proyecto), para tal efecto se propone el caso de estudio que muestra el problema de parentesco familiar.

El apéndice ?? contiene información de XML y otros lenguajes de marcado, además incluye diversas tecnologías usadas para manipular dicho lenguaje.

Capítulo 2

Bases activas de documentos XML

2.1. Introducción

Recientemente se ha extendido el uso de las bases de documentos XML (BDX), como medio de almacenamiento de datos, debido a que ofrecen ventajas que los Sistemas Gestores de Bases de Datos (SGBD) no ofrecen. En este capítulo se describen las ventajas que se obtienen al usar XML para almacenar y compartir datos. También aquí se definen los conceptos básicos de las reglas ECA y se mencionan algunos trabajos relacionados. Algunas herramientas que facilitan el uso de XML se describen en el apéndice B.

2.2. XML dentro de los lenguajes de marcado

De acuerdo con [SM] podemos definir el marcado como cualquier medio para hacer la interpretación de un texto explícito. Por lenguaje de marcado entendemos un conjunto de convenciones de marcado usadas juntas para codificar textos [vdG97]. Dentro de los lenguajes de marcado más conocidos, se destacan SGML, HTML, XML y XHTML, en el apéndice B los describimos brevemente así como la relación que existe entre ellos.

2.2.1. XML

XML (Extensible Markup Language) es un dialecto simple de SGML que se definió suprimiendo algunas características y opciones de la especificación de SGML. El objetivo de XML es recuperar, recibir y procesar documentos SGML de la Web [Kil]. La característica fundamental de XML es que las etiquetas de este lenguaje no están predefinidas, sino que cada aplicación debe definir sus propias etiquetas. XML usa una DTD o un esquema XML para describir los datos. De esta manera, XML está diseñado para ser autodescriptivo con una DTD o un esquema.

Hay varias ventajas de publicar el contenido de base de datos como XML, algunas de ellas se muestran en seguida[SR01]:

1. Separa claramente el contenido de la presentación. Para hacer esta separación, XML se encarga del contenido únicamente, mientras que la presentación se maneja por separado (por ejemplo CSS, XSL, o XHTML).
2. Soporta validación. La validación permite construir documentos gramaticalmente correctos.
3. Es autodescriptivo. Usando DTD o Schema La salida XML incluye su propia información de estructura, es decir, para alguien que entiende las etiquetas usadas, la información es autodescriptiva.

4. Permite guardar parte del formato abierto. Con los esquemas XML si se insertan nuevas etiquetas, los sitios equipados para usar la información nueva pueden hacerlo, mientras que los analizadores gramaticales de los otros sitios no tienen problema (no hacen caso de ellas).
5. Provee ayuda indirectamente para la distribución de datos. Esto se debe a que soporta algunos mecanismos para la invocación de funciones remotas a través del Web. Por ejemplo, el Simple Object Access Protocol (SOAP) usa XML y HTTP como un mecanismo de invocación de métodos.
6. Permite intercambiar datos entre sistemas incompatibles. En el mundo real, los sistemas informáticos y bases de datos frecuentemente contienen datos en formatos incompatibles. Uno de los desafíos que requiere mucho tiempo por parte de los desarrolladores es intercambiar datos entre tales sistemas sobre la Internet. La conversión de los datos a XML puede reducir enormemente esta complejidad al crear datos que se pueden leer usando muchos tipos diferentes de aplicaciones.
7. Se puede usar para datos heterogéneos. XML proporciona una sintaxis neutral para describir datos que tienen la estructura de grafos en forma de datos anidados. Ya que se puede transformar estructuras de datos diversas en tales grafos, XML ofrece un modo de representar estructuras de datos heterogéneas. Añadiendo DOM (y quizás un lenguaje de consulta) proporciona las operaciones necesarias para tener acceso a estas estructuras de datos.
8. Se puede usar para datos estructurados. Cuando una fuente o receptor maneja datos organizados jerárquicamente (p.ej., para mensajes formateados), XML ayuda en la reestructuración de la información a formatos relacionales y jerárquicos. Para representar estructuras de datos, XML proporciona tanto representación (con sus lenguajes de consulta) como capacidad para manipular/transformar. Esto permite a un receptor volver a reensamblar la misma estructura de datos que le fue enviada.
9. Proporciona un mecanismo para adjuntar metadatos. Los metadatos son literalmente datos sobre datos y se adjuntan a los atributos de los esquemas tanto de la fuente como del objetivo. Los metadatos se requieren cuando se tienen representaciones heterogéneas de atributo y semántica entre la fuente y el destino.
10. Facilita conectividad universal de ambientes de Web. Significa que se pueden hacer definiciones de elemento estándar fácilmente, bibliotecas de función de conversión y otros recursos que promueven la interoperabilidad.
11. Reduce los gastos de desarrollo. Debido a la omnipresencia de XML, los constructores de herramientas se benefician de un mercado grande, por ello las herramientas XML (software libre y comercial) ofrecen flexibilidad a bajo costo. Otra razón por la que reduce costos de desarrollo es por que cuando se comparten datos entre SGBDs diferentes, es necesario el uso de software intermedio para reconciliar las diferencias entre fuentes de datos. Sin embargo si se usa XML no se requiere usar software intermedio.

2.3. Reglas ECA en sistemas de bases de datos

En esta sección se hace una descripción de la importancia de las reglas ECA, la principal fuente de información es [\[HW93\]](#).

2.3.1. Introducción a la reglas ECA

Los sistemas de base de datos son pasivos pero si un sistema de base de datos también provee mecanismos para crear y procesar reglas de producción, se vuelve activo. El paradigma de reglas de producción de Inteligencia Artificial (IA) se ha modificado para el contexto de bases de datos activas. Algunos de los beneficios que se obtienen al usar reglas de producción son: asegurar la integridad, monitorear el acceso y evolución de los datos, el mantenimiento de datos derivados, reforzar esquemas de protección, mantener versiones históricas, entre otras.

Algunos ejemplos de sistemas de bases de datos son: Ariel, POSTGRESS, RPL, Startburst (relacionales), HiPAC y Ode (los últimos dos son orientadas a objetos). En un sistema de bases de datos relacional todos los datos se almacenan en tablas (o relaciones). Asociado con cada tabla hay un número de columnas (o atributos) y cada tabla contiene cero o más tuplas (o registros). En un sistema orientado a objetos, todos los datos son almacenados en objetos.

2.3.2. Lenguajes de reglas de bases de datos

En IA las reglas tienen la forma: *patrón* \rightarrow *acción*, donde *patrón* es una condición o un predicado. A tales reglas se les llama *basadas en patrones*. La regla se activa (su término en inglés es *trigger*) cuando el patrón coincide (*match*) con los datos que se encuentran en la *memoria de trabajo*, entonces, la acción modifica la memoria de trabajo, posiblemente de acuerdo a los datos que han coincidido con el patrón. La diferencia más obvia entre los lenguajes de reglas de IA y los lenguajes de reglas de bases de datos es que en muchos lenguajes de reglas de bases de datos la activación del evento o eventos se especifica explícitamente. Por ello a estas se les denomina *basadas en eventos*.

Especificación de eventos

Un evento es un cambio observable en la estructura o el contenido de información de un sistema. La posibilidad de observar al evento depende tanto de la entidad bajo observación como del observador. Así mismo, el origen de la alteración del sistema puede clasificarse de acuerdo al tipo de modificación que induce, agrupadas aquí en cuatro categorías:

- **En la estructura de información.** Además de modificar los objetos mediante su creación, destrucción o actualización, también existen modificaciones a nivel meta-objeto cuando cambia la estructura de los objetos mismos al adicionarse, eliminarse o modificarse cualquiera de sus componentes. Por ejemplo, en una tabla de una base de datos, la inserción de un nuevo registro a la base de datos es una alteración al nivel de objeto (i.e., colecciones de objetos), pero la adición de nuevos campos es una alteración al nivel de meta-objeto (i.e., colecciones de atributos).
- **En el contenido de información.** Los objetos no modifican su estructura pero si su contenido. Por ejemplo, al modificar el contenido de uno de los campos de un registro en una tabla.
- **En el comportamiento interno.** De acuerdo a los mecanismos de visibilidad que ofrezca el sistema, la actividad interna de un sistema puede percibirse a través de los mensajes que se intercambian o de las invocaciones a operaciones que se realizan. Las operaciones internas pueden clasificarse a su vez en varias subcategorías como persistencia, seguridad, localidad, contingencia (excepciones y compensaciones), transaccional (atomicidad, consistencia, aislamiento y durabilidad), y temporalidad (tiempo absoluto y relativo, eventos periódicos y recurrentes, intervalos), entre otras.

- **En el comportamiento externo.** En este caso, el cambio ocurre fuera de las fronteras del sistema pero su observación tiene consecuencias en el interior del sistema. Por ejemplo, el inicio o el término de procesos externos, el fallo de un sistema computacional en sistemas tolerantes a fallos, la medición de un parámetro ambiental como la presión, humedad o temperatura.

Los eventos elementales generalmente se agrupan en patrones característicos reconocibles por la estructura estructural-temporal que poseen. De acuerdo a su estructura, los eventos compuestos se pueden clasificar de la siguiente forma:

- **Estructurales:** Los eventos ocurren en un espacio o una región bien definida de un sistema, independientemente del orden o del tiempo en el que sucedieron (no existen relaciones temporales antes-de entre ellos). Entre los conectivos más importantes podemos destacar:
 - **Conjunción** $all(e_1, e_2)$, ambos eventos, e_1 y e_2 , han ocurrido,
 - **Disyunción** $any(e_1, e_2)$, cualquiera de los eventos, e_1 o e_2 , ha ocurrido,
 - **negación** $not(e)$, e_1 evento e no ha ocurrido.

Estos conectivos extienden sus definiciones naturalmente a listas no vacías de eventos.

- **Temporales:** Los conectivos temporales incorporan una relación de orden total (antes-de y simultáneamente-con) entre ellos sin importar el espacio o la región del sistema donde ocurrieron. Algunos de los conectivos son:
 - **antes** $before(e_1, e_2)$, el evento e_1 ocurre antes que el evento e_2 ,
 - **simultáneamente** $simultaneous(e_1, e_2)$, el evento e_1 ocurre simultáneamente con e_2 ,
 - **después** $after(e_1, e_2)$, el evento e_2 ocurre después del evento e_1 ,
 - **durante** $during(e_1, e_2, e_3)$, el evento e_1 ocurre en el segmento comprendido entre e_2 y e_3 , es decir, e_1 ocurre antes que o simultáneamente con e_2 pero antes que e_3 ; cuando los eventos e_2 y e_3 son temporales, este conectivo corresponde con la noción usual de intervalo,
 - **siguiente** $next(e_1, e_2)$, el único evento observable después de ocurrir e_1 es e_2 , Aunque algunos de estos conectivos se pueden expresar en términos de los conectivos antes y simultáneamente, su definición simplifica considerablemente las reglas de composición de eventos. Por otra parte, los conectivos se pueden aplicar a listas no vacías de eventos recurriendo al uso de los conectivos estructurales. Por ejemplo, el evento compuesto $before(all(es), e)$ se observa cuando todos los eventos que aparecen en la lista es ocurren antes que el evento e . De la misma forma, el evento compuesto $not(during(any(es), e_1, e_2))$ se observa cuando ninguno de los eventos indicados en la lista es ocurren en el segmento delimitado por e_1 y e_2 .

La administración de eventos establece los conceptos fundamentales de historia y de horizonte para su tratamiento formal. La *historia de los eventos* indica la cronología (indexamiento temporal) de los eventos. La historia asocia a cada evento un número ordinal que representa al tiempo. La resolución del reloj (frecuencia de muestreo) determina el refinamiento con la que los eventos se pueden situar en la recta numérica. El *horizonte de los eventos* es el conjunto de eventos elegibles para formar eventos compuestos. Entre los tipos de horizontes que podemos definir se encuentran: *horizonte reciente*, en donde se consideran aquellos eventos posteriores a la ocurrencia de algún evento reciente; *horizonte lejano*, en donde se consideran aquellos eventos

anteriores a algún evento dado; *horizonte medio*, en donde se consideran aquellos eventos situados en un segmento delimitado por dos eventos primitivos dados y *horizonte infinito*, en donde se consideran todos los eventos observados desde el inicio.

La estructura e información asociada con el evento se asocia generalmente a este, de manera que sea manipulable por la aplicación. De esta manera, si se considera que un mensaje debe inducir un comportamiento determinado en un sistema, entonces es necesario conocer el contenido de dicho mensaje para producir el comportamiento esperado.

Aunque la estructura sintáctica y la lectura declarativa de los eventos que hemos dado puede sugerir que se trata de operaciones lógicas ordinarias, es necesario reconocer sus diferencias. Entre las más notorias se encuentran:

- **Los eventos conllevan comportamiento paralelo asíncrono.** El comportamiento paralelo asíncrono implica que los eventos concurrentes ocurren independientemente unos de otros. En general, la ocurrencia de los eventos es impredecible y, por lo tanto, incontrolable (ausencia de un reloj o control global). La falta de control da lugar a que no se pueda utilizar ninguno de los paradigmas estrictamente computacionales conocidos para describirlos. En otras palabras, no se puede escribir un programa que anticipe la ocurrencia de eventos; a lo más, solamente se puede aspirar a observar aquellos eventos o patrones de eventos de interés. Como consecuencia se deben abandonar los modelos puramente computacionales para adoptar modelos interactivos.
- **Los eventos caracterizan a los sistemas abiertos.** El comportamiento paralelo asíncrono es el comportamiento característico de los sistemas abiertos ya que no restringen el ordenamiento relativo de los eventos generados u observados en los sistemas componentes. Los sistemas abiertos únicamente definen la estructura de los eventos observables (interfaz de programación) pero no prescriben en general las acciones a tomar como consecuencia de su observación. El desacoplamiento de la interfaz de programación del comportamiento interno del sistema extiende considerablemente el poder expresivo de un modelo basado en eventos sobre un modelo estrictamente computacional.
- **Los eventos introducen problemas potenciales.** La descomposición y el desacoplamiento de un sistema guiado por eventos introducen fuentes adicionales de anomalías y errores debido a la introducción del medio de comunicación en donde se observan los eventos. La pérdida, el retraso o la alteración del evento observado puede causar problemas que pueden no tener solución.

Al reconocer que los eventos son de naturaleza más compleja que las condiciones lógicas, se deben reconocer las dificultades inherentes al modelo de programación que integre ambos conceptos.

Los eventos más comunes en lenguajes de reglas de producción en bases de datos son modificaciones a la base de datos. En los sistemas de bases de datos relacionales, las modificaciones a la base de datos se hacen mediante los comandos de inserción, borrado o actualización; mientras que en los sistemas de bases de datos orientados a objetos, estas modificaciones se hacen mediante invocación a métodos. Algunos lenguajes de reglas de producción de bases de datos permiten también que se activen reglas mediante recuperación de datos. Existen además los eventos **temporizados (timing events)** que son aquellos que se activan mediante el reloj de la computadora, ya sea a una hora del día o en intervalos de tiempo. A continuación se muestra la forma en que manejan los eventos tanto los sistemas relacionales como los sistemas orientados a objetos:

- **Sistemas relacionales:** en Ariel y Starburst los eventos son inserciones, eliminaciones o actualizaciones. En Ariel, la parte del evento se puede omitir, por ello las reglas son basadas

en patrones. POSTGRES permite que los eventos sean operaciones de recuperación de datos. En RPL las reglas son puramente basadas en patrones, así que no se especifican los eventos.

- **Sistemas orientados a objetos:** en Ode las reglas son puramente basadas en patrones. En HiPAC son operaciones a la base de datos (recuperación, inserción, eliminación, actualización) mediante invocaciones a métodos.

Especificación de la condición

La condición es la parte de la regla que especifica un predicado o consulta sobre los datos en la base de datos y si la condición se satisface, la acción de la regla se ejecutará. En el caso en que la condición es una consulta, se dice que la condición se satisface si la consulta devuelve datos. Existen casos en los que la condición se puede omitir, en tal caso se espera sólo que ocurra un evento para que la acción de la regla se ejecute.

En POSTGRES, Starburst y Ariel las condiciones son predicados arbitrarios sobre el estado de la base de datos. En HiPAC y RPL las condiciones son conjuntos de consultas en la base de datos. En Ode la condición es un predicado sobre el valor del objeto en el cual se invocó el método.

Las condiciones son expresiones lógicas que se construyen sobre las constantes y los conectivos lógicos usuales, a los cuales se incorporan los operadores relacionales definidos sobre diferentes tipos de datos y aplicados al contenido de información tanto del sistema como de los eventos. Para distinguir entre ambos, designaremos como *contexto* a la información que contiene el sistema. Así el contexto establece el marco de evaluación de una condición. Entre las restricciones más importantes se encuentran aquellas que se designan bajo el nombre de restricciones de integridad (*integrity constraints*) que aseguran la consistencia de la información. Las restricciones de integridad pertenecen a un grupo de propiedades de correctitud (*correctness*) conocidas como de seguridad (*safety*) y se refieren a aquellas propiedades que se cumplen en todos los estados del sistema. Debido a las diferencias fundamentales entre eventos y condiciones, es necesario reconocer que la evaluación de una condición puede dar diferentes resultados a lo largo de la formación de un evento compuesto. Por lo tanto, cuando se definen condiciones sobre eventos, la verificación de restricciones de integridad debe indicar además el momento en el que se debe realizar la evaluación de la condición. Por ejemplo, en un catálogo que contiene registros cuya llave primaria ocurre en la tabla exactamente una vez, la operación de actualización `before(delete r_1 , insert r_2)` haría que la restricción de integridad no se cumpliera cuando la evaluación se realiza después de eliminar al registro r_1 pero antes de insertar su actualización r_2 .

Especificación de la acción

La acción especifica las operaciones que se realizan cuando se activa una regla y su condición se satisface. En los diversos sistemas de bases de datos, las acciones varían desde operaciones de inserción, borrado y actualización (p.ej. en RPL), e incluso pueden ser secuencias arbitrarias de comandos de recuperación o modificación (p.ej. Ariel, POSTGRESS y Startburst), o también pueden especificar *rollback*.

Las acciones se definen sobre las operaciones básicas que exponen los componentes de un sistema y sobre los cuales se pueden agrupar en estructuras de control clásicas como la composición secuencial, paralela, alternativa, condicional e iterativa. Por ejemplo, en una base de datos las operaciones fundamentales son inserción, eliminación, actualización y selección de registros. Al igual que en las condiciones, el contexto determina el marco en el que se realizan las acciones, pero

a diferencia de ellas, las acciones modifican el contexto. De acuerdo al modelo de programación, las acciones pueden observarse por su invocación o por el efecto que producen ya que generan cambios en la estructura y la información del sistema. En general, los eventos relacionados con invocaciones son más flexibles y expresivos que aquellos relacionados con modificaciones. Por ejemplo, para asegurar el cumplimiento de las restricciones de integridad de una base de datos, es preferible determinar si la inserción de un registro preserva la consistencia de las tablas antes de insertarlo que restaurar las tablas a un estado consistente después de haberlo insertado.

2.3.3. Vinculación entre eventos, condiciones y acciones

Para comunicar las partes de evento, condición y acción de una regla se requiere que haya un enlace o vínculo entre ellas. Para hacer dicha vinculación, en algunos lenguajes como HiPAC la activación del evento se hace enviando parámetros a los cuales se puede hacer referencia en la condición y acción de la regla. En algunos sistemas de bases de datos como Ariel, se emplean *variables de tupla* que permiten lograr esta vinculación, por ejemplo la palabra clave `previous` hace referencia al valor anterior de la tupla actualizada. En POSTGRESS se emplean las palabras clave `new` y `old` para hacer referencia al valor actual y anterior respectivamente.

En otros DBMSs como Starburst para hacer el enlace, se emplean las *tablas de transición*. Las tablas de transición son tablas lógicas y se usan igual que las tablas de la base de datos. En tiempo de ejecución, la tabla de transición `inserted` contiene las tuplas que al insertarse, activaron la regla. La tabla de transición `deleted` contiene las tuplas que al ser eliminadas, activaron la regla. Las tablas de transición `new-updated` y `old-updated` contienen los nuevos y antiguos valores respectivamente de las tuplas que se actualizaron y activaron la regla.

2.3.4. Ordenamiento de reglas

Se denomina *resolución de conflictos* a la elección de qué regla ejecutar cuando se activan múltiples reglas. En muchos DBMSs activos esta elección se hace más o menos arbitrariamente y en otros no se ha contemplado. Los DBMSs resuelven de diversas formas este conflicto, algunos como POSTGRESS y Ariel asignan *prioridades numéricas* a cada una de las reglas, en otros como Starburst la reglas se encuentran *parcialmente ordenadas* y hay otros (por ejemplo HiPAC) que ejecutan la reglas de forma *concurrente*.

2.3.5. Semánticas de procesamiento de reglas

La semántica del procesamiento de reglas determina cómo tomarán lugar las reglas de procesamiento en tiempo de ejecución una vez que un conjunto de reglas se definió. También determina como interactúan las reglas con las operaciones y transacciones arbitrarias de la base de datos emitidas por usuarios y programas.

De diversas alternativas existentes para la ejecución de reglas, se describe una variación del `recognize-act cycle` (usado en la mayoría de los sistemas en AI). Funciona de la siguiente forma: se hace una comparación (*match*) inicial ejecutando la condición de la regla para determinar qué reglas se activan y para cuales instancias. Cada tupla producida por la consulta en una condición de regla es una concretización para esa regla. El conjunto de instancias de regla activadas se llama *conjunto conflicto*. Estos pasos pueden formar un ciclo, que se detiene hasta que las condiciones de regla no producen tuplas. Dentro del ciclo se usa una *estrategia de resolución de conflictos* para escoger una regla y se ejecuta la acción de la regla para cada tupla producida por la condición, luego se hace una comparación (*match*), para verificar si el ciclo sigue o se detiene. Disparar (*firing*) la regla seleccionada para todas las instancias se conoce como *saturación*.

Tolerancia a fallos

Una regla en bases de datos puede generar errores durante su ejecución, algunas razones son: los datos dependen de una tabla que se borró, los privilegios de acceso a datos se han revocado, las transacciones que se ejecutan concurrentemente generan un abrazo mortal etc. Muchos sistemas de reglas en bases de datos manejan los errores durante el procesamiento abortando la transacción actual. Sin embargo en el caso de condiciones de error producidas por las acciones de las reglas, otras alternativas son: terminar la ejecución de aquella regla y seguir el procesamiento de las reglas, volver al estado que precede el procesamiento de regla o comenzar de nuevo el procesamiento de regla.

2.3.6. Arquitectura de programación

Los sistemas de bases de datos activas, deben soportar todas las características que proveen los sistemas de bases de datos convencionales incluyendo: definición de datos (para describir el formato de los datos), manipulación de datos (para hacer consultas y modificaciones), manejo de almacenamiento, manejo de transacciones, control de concurrencia, y tolerancia a fallos (*crash recovery*). Además deben proveen mecanismos para la activación de reglas y la verificación de condiciones para cada ejecución de las acciones.

Evaluación de condiciones (Condition testing)

En muchos sistemas de bases de datos, incluyendo HiPAC, Ode, RPL y Starburst, las condiciones de las reglas se prueban durante el procesamiento de reglas, esto se hace consultando la base de datos. Lo anterior puede hacer que el desempeño del sistema disminuya. En POSTGRES, se ponen marcadores en las tuplas que satisfacen la condición de la regla, de esta manera no es necesario probar las condiciones durante el procesamiento de reglas. En Ariel el mecanismo de probar las condiciones usa un algoritmo llamado A-TREAT, que es descendiente de TREAT, un algoritmo que se usa en sistemas de reglas de producción en memoria principal.

Ejecución de las acciones

En todos los sistemas de reglas en bases de datos, las acciones de las reglas se ejecutan enviando operaciones al procesador de consultas. Sin embargo muchos sistemas requieren algunos mecanismos adicionales para vincular los datos que activan una regla con las operaciones en la acción de la regla. En Ariel, cuando una regla se calendariza para su ejecución, las tuplas cuya modificación activó la regla se almacenan en el nodo *p* de esa regla. Cuando se ejecuta un comando en la acción de la regla, las tuplas para la tabla de activación de la regla se derivan explorando el nodo llamado *p* en lugar de acceder a la tabla en la base de datos. En POSTGRES, el problema de vinculación es relativamente simple dado que las reglas se activan por cambios a una única tupla. En RPL el procesador de reglas determina qué tuplas coinciden (*match*) con una regla haciendo consultas a la base de datos; se envían identificadores de estas tuplas como parte del comando para ejecutar la acción de la regla. En Starburst, los datos de activación se acceden mediante tablas de transición, mismas que se hacen durante el procesamiento de consultas. En HiPAC los datos de activación se pasan a la acción de la regla usando parámetros.

Componentes principales de la arquitectura

La arquitectura describe los componentes y las interacciones que realizan en la administración de la ejecución de un conjunto de reglas. Los componentes más importantes son:

- **Observador** detecta los eventos elementales que se producen en el interior o el exterior del sistema; una vez identificados, los registra en la historia,
- **Reconocedor** identifica eventos compuestos al reconocer el patrón que forman eventos registrados en la historia; una vez identificados, registra los eventos compuestos en la historia,
- **Seleccionador** elige las reglas de acuerdo al tipo de eventos que las activan; una vez seleccionadas, produce instancias de ellas,
- **Evaluador** determina en cuáles instancias se verifica su condición de acuerdo al contexto; en caso de existir varias instancias de reglas cuya condición se verifique, entonces se tiene un conjunto de reglas en conflicto,
- **Mediador** de conflictos resuelve el conflicto entre las instancias de las reglas usando criterios como antigüedad, prioridad, justicia y ordenamiento lexicográfico, entre otros,
- **Despachador** establece el orden de la agenda con que las instancias seleccionadas serán ejecutadas, y
- **Ejecutor** realiza las acciones de la instancia de la regla; en caso de que las acciones modifiquen el contexto de manera que hagan inválida la condición de las instancias seleccionadas, entonces dichas instancias dejan de considerarse para su ejecución.

Las interacciones entre estos componentes en general no se pueden realizar en forma secuencial por lo que se pueden considerar únicamente acoplamientos entre los pares evento-condición y condición-acción. Los modos de acoplamiento más comunes son:

- **Inmediato.** En donde la condición (acción) se evalúa (ejecuta) inmediatamente después de observar (evaluar) al evento (condición) ya que ambas se encuentran en el mismo proceso,
- **Diferido.** En donde la condición (acción) se evalúa (ejecuta) después de observar (evaluar) al evento (condición) (aunque no necesariamente de inmediato) ya que el proceso del primero causa la reanudación del proceso del segundo,
- **Desacoplado** En donde la condición (acción) se evalúa (ejecuta) en un proceso distinto del que observa al evento (condición).

2.4. Trabajo Relacionado

2.4.1. Servicios Web y Repositorios Activos de documentos XML

Se han hecho algunos trabajos en los que se combinan las tecnologías de servicios Web y repositorios activos de documentos. En [BCP01] los autores proponen el uso de reglas activas para desarrollar servicios Web. En ese trabajo usan XSLT (un lenguaje basado en patrones para publicar documentos XML) o Lorel (un lenguaje de consultas para documentos XML) para definir las reglas ECA. Dada la naturaleza de ambos lenguajes, cada regla se compone de dos partes: el evento y la condición-acción, esta es una diferencia con nuestro trabajo ya que nosotros manejamos cada parte de la regla (eventos, condiciones y acciones) de forma separada. Otra diferencia importante es que en nuestro trabajo las reglas se pueden activar dependiendo de las respuestas de servicios Web, mientras que en el trabajo presentado en [BCP01] las reglas se activan únicamente cuando se perciben cambios en el repositorio de documentos. Por último, podemos mencionar que en esta tesis se usa la inferencia lógica para hacer búsquedas en los repositorios de documentos XML, sin embargo en [BCP01] esta capacidad deductiva no fué incorporada.

2.4.2. Uso de servicios reactivos con reglas activas en repositorios XML

En [BCP02] los mismos autores de [BCP01] proponen el uso de reglas activas en repositorios XML para implementar la tecnología *push*. En este artículo hacen uso de tecnologías conocidas como: DOM, XQuery y SOAP. En [BCP02] si se establece la diferencia entre la condición y la acción de la regla. La acción de la regla se refiere a la entrega de información exclusivamente. En nuestro trabajo, la acción de la regla puede ser desde consulta y actualización en bases de datos relacionales, hasta la invocación de servicios Web. Como en el caso anterior, en [BCP02] la capacidad deductiva tampoco fué incorporada.

Las reglas operan en documentos XML en reacción a eventos de actualización que ocurren en el repositorio. Las reglas supervisan los eventos que ocurren en los sistemas remotos y notifican a consumidores interesados en la información. Cada regla actúa como un servicio Web independiente; un protocolo B2B regula la instalación remota de las reglas en el servidor, el facilitador de reglas propone el protocolo y el repositorio remoto lo acepta, esta negociación sigue un simple contrato de instalación.

Para manejar propiedades de las reglas activas (como terminación y confluencia), en el mencionado trabajo emplean reglas que sólo tienen la capacidad de notificar a usuarios remotos y por ello, no pueden activarse entre si, de esta manera se garantiza la *terminación*. El orden de notificación a los usuarios se determina usando políticas de resolución de conflictos.

Para el protocolo de negociación y específicamente para el intercambio entre el facilitador de reglas y el repositorio XML se emplea SOAP debido a que el uso de SOAP como un mecanismo genérico de invocación de servicios Web hace la solución flexible y portable.

Reglas ECA

El interés se enfoca en los *mutating events* y los describen como los eventos que se generan cuando un nodo (elemento o atributo) se modifica (inserción, borrado o modificación) y se necesita monitorear esta modificación. Cada vez que ocurra una modificación, se genera la instancia correspondiente de evento y se asocia con el nodo modificado. Para la detección de eventos ocupan un componente de la interfaz DOM llamado *Event Model* que se introdujo en la segunda especificación. La detección de eventos se hace mediante *event listeners* asociados con nodos DOM, estos *event listeners* detectan los eventos que ocurren en los nodos en los cuales están asociados o en sus descendientes.

La condición se expresa como una consulta XML escrita en XQuery. Para el manejo de las condiciones, una característica importante es la presencia de un mecanismo de comunicación entre la condición y la parte del evento, esto se requiere para tener una forma de referirse a los nodos en los cuales ocurren los eventos. Para lo anterior, se usan las variables predefinidas *new* y *old* que representan los valores actual y anterior respectivamente de los nodos modificados.

La parte de la acción especifica un método SOAP. Se restringe este método SOAP para implementar la llamada a un sistema de entrega de mensajes, que transferirá la información a recipientes específicos. Las reglas no tienen prioridades por lo tanto la ejecución de las reglas no tiene *confluencia*. La implementación del canal de comunicación entre la condición y la acción se hace permitiendo el uso de las variables de la condición en la acción.

La arquitectura

La arquitectura general que se maneja consiste en tres actores principales: el proveedor de servicios (*Service Supplier*), el intermediario de servicios (*Service Reseller*) y el facilitador de reglas (*Rule Broker*). El proveedor de servicios entrega bienes y servicios descritos por un Servidor XML

que internamente soporta un motor de ejecución de reglas de XML. Las reglas monitorean eventos (como la nueva disponibilidad de un servicio) y luego notifican al intermediario de servicios. El intermediario de servicios es el recipiente de mensajes acerca de nuevos servicios; y típicamente actúa recíprocamente con los clientes que están interesados en la compra de bienes o servicios específicos. El facilitador de reglas actúa como un intermediario; recibe la información sobre los servicios que se buscan sobre la Internet e instala reglas en los sitios del proveedor de servicios. Para lo cual, debe poner las reglas en un formato conveniente y luego instalarlas remotamente.

Proponen una interfaz B2B entre el facilitador de reglas y el proveedor de servicios, basada en cuatro primitivas SOAP (`Connect`, `Subscribe`, `Unsubscribe` y `Disconnect`) que el facilitador de reglas invoca y el Servidor XML soporta. Las primitivas se implementan como invocaciones SOAP al servidor XML. Ellos denominan esta interfaz el *Protocolo de Envío de Regla* (*Rule Submission Protocol*). Se debe extender el Servidor XML para soportar el protocolo mencionado anteriormente y ejecutar reglas de XML.

2.4.3. Análisis y optimización de Reglas ECA en XML

En [BPW01] se definió un lenguaje simple para especificar reglas ECA en repositorios XML. Además se investigaron los métodos para analizar y optimizar reglas ECA. Una característica importante de dicha publicación es que a diferencia de otros trabajos, ahí sí explican la sintaxis y la semántica del lenguaje propuesto. A continuación se resume la sintaxis y semántica de las reglas usadas en dicho artículo. En dicho artículo los autores resaltan el hecho de que su trabajo es el primero en que se combinan las reglas ECA con los datos XML. Ellos afirman que determinar la activación de reglas de ECA es más compleja en XML que para bases de datos relacionales, porque la determinación de los efectos de las acciones de regla no es simplemente un asunto de correspondencia de los nombres de relaciones actualizadas con eventos potenciales o con los cuerpos de condiciones de regla. En cambio, las asociaciones son más implícitas y se requieren comparaciones semánticas entre los conjuntos de expresiones de ruta.

Sintaxis de las reglas ECA

Un *evento* es una expresión de la forma: `INSERT e` o `DELETE e` , donde e es una evaluación de expresión XPath a un conjunto de nodos que son las raíces de subdocumentos nuevos (en caso inserción) o borrados. La variable `$delta` definida por el sistema, está ligada a las raíces de los subdocumentos nuevos o borrados y está disponible para usarla tanto en la parte de condición como en la acción de la regla.

La *condición* puede ser la constante `TRUE` o una conjunción de expresiones XPath. Se evalúa en cada documento XML que se ha modificado por un evento de la forma especificada en la parte de evento de la regla. Si la condición hace referencia a la variable `$delta` entonces se evalúa una vez para cada instanciación de `$delta` para cada documento. De otra manera, se evalúa sólo una vez por cada documento.

La *acción* es una secuencia de una o más acciones: `accion1; ... ; accionn`. Estas se ejecutan en cada documento XML que se ha modificado por un evento de la forma especificada en la parte de evento de la regla y que la consulta de la condición de la regla se evalúa como verdadera. A este conjunto se le llama *documentos candidatos*. Se dice que la regla se dispara si el conjunto de documentos candidatos es no vacío.

Cada acción es una expresión de la forma:

```
INSERT  $r$  BELOW  $e$  [BEFORE | AFTER  $q$ ]
```

DELETE *u* BELOW *e*

donde: *r* es una expresión XQuery, *e* y *u* son expresiones XPath, y *q* es un calificador XPath.

Semántica de ejecución de las reglas ECA

Se especifica la semántica de la ejecución de las reglas usando el pseudocódigo que se muestra abajo. La entrada para la ejecución es una *base de datos XML db*, y un *plan de actividades s*. El plan consiste de una lista de *actualizaciones* que se deben ejecutar en esa base de datos. Cada actualización es un par (*a*, *docsAndDeltas*), donde *a* es una acción de la parte de acciones de alguna regla, y *docsAndDeltas* es un conjunto de pares (*d*, *deltas*) donde *d* es un identificador de un documento candidato sobre el que debe aplicarse y *deltas* es un conjunto de instancias para la variable *\$delta* con respecto al documento *d*, se llama a ese conjunto de instancias el conjunto de *deltas*. Las reglas se identifican por números 1..*noDeReglas* en orden de prioridad decreciente (se asume que dos reglas no tienen la misma prioridad). Y se asume que el plan que concretiza la ejecución de la regla ECA consiste de una acción de una de las reglas, y un conjunto inicial de documentos y *deltas* sobre los cuales se aplicará esta acción, por ejemplo el plan inicial es una lista con un elemento de la forma [(*a*, *docAndDeltas*)]:

```
While s != [ ] do {
  (a, docAndDeltas) := head s;
  s                 := tail s;
  (deltas, db)      := updateDB db (a, docAndDeltas);
  for i := noOfRules downto 1 do {
    docAndDeltas := candidateDocs i deltas db;
    if docAndDeltas != {} then
      for j := noOfActions[i] downto 1 do
        s := (action[i , j], docsAndDeltas):s
  }
}
```

La función `head` devuelve el primer elemento de la lista y la función `tail` devuelve la lista menos su primer elemento. La función `updateDB` ejecuta la acción *a* en la cabeza del plan *s* sobre el conjunto de documentos especificados en *docsAndDeltas*. Si *a* hace referencia a *\$delta*, se hace una actualización para cada instancia en cada documento candidato. `updateDB` devuelve un par (*deltas*, *db*), *db* es la base de datos resultante de esta actualización, *deltas* es un arreglo bidimensional tal que *deltas*[*i*, *d*] es el conjunto de deltas correspondiente a la parte de evento de la regla *i* con respecto al documento *d*. La función `candidateDocs` determina el conjunto de documentos candidatos para la regla *i* con respecto a la actualización que se acaba de ejecutar en la base de datos por `updateDB` junto con un conjunto de instancias de la variable *\$delta* para cada documento candidato. Esto se hace evaluando la condición de la regla *i* en todos los documentos *d* para los cuales *deltas*[*i*,*d*] no es vacío. Se deben considerar dos casos para cada documento:

1. La variable *\$delta* ocurre en la condición de la regla *i*:

La condición se evalúa para cada elemento de *deltas*[*i*,*d*], y se determina el subconjunto *D* de *deltas*[*i*,*d*], para el cual la condición evalúa a verdadera. Si *D* es no vacío, entonces se regresa *d* como documento candidato y *D* es asociado con el como el conjunto de instancias para *\$delta* con respecto a *i* y *d*. De otra forma, no se devuelve a *d* como documento candidato.

2. No hay ocurrencia de la variable $\$delta$ en la condición de la regla i :

La condición solo se evalúa para un documento d . Si evalúa a verdadera, se devuelve d , asociado con el conjunto de $deltas[i, d]$ como el conjunto de instancias para $\$delta$ con respecto a i y d . De otra forma, no devuelve d como documento candidato.

$noOfActions[i]$ es el número de acciones en la parte de acción de la regla i , y $actions[i, j]$ es la j -ésima acción de la regla i . La sentencia `for j := noOfActions[i] downto 1 do ...` asegura que las acciones de una regla dada se ponen en el orden correcto en el plan.

Se asume que el modo de acoplamiento es **inmediato**, quiere decir que las acciones de las reglas que se han disparado como consecuencia de la actualización reciente se colocan a la cabeza del programa actual.

Análisis del comportamiento de las reglas ECA

Para analizar reglas ECA, en [BPW01] se pone especial énfasis en la terminación de la ejecución de las reglas. Se dice que un conjunto de reglas tiene terminación si para cualquier evento inicial y para cualquier estado inicial de la base de datos, la ejecución de las reglas termina. Se consideran que no es suficiente traducir los datos XML y reglas ECA a una forma relacional para su análisis, sino que se necesita desarrollar nuevas técnicas para analizar directamente. Ellos proponen 3 técnicas:

- Grafos de habilitación (*Triggering graphs*): para reglas ECA XML.
- Grafos de activación (*Activation graphs*): para reglas ECA XML.
- Interpretación abstracta de la ejecución de la regla.

En esta tesis no se analiza el comportamiento de reglas ECA, por tal razón no se dan mas detalles de las técnicas mencionadas en [BPW01].

2.5. Resumen

En este capítulo se describieron las ventajas que se obtienen al usar XML como medio para almacenar y compartir información. Algunas de las ventajas mas sobresalientes son: que XML soporta validación, es auto descriptivo, facilita la descripción de datos estructurados, reduce gastos de desarrollo, entre otras.

Se vió que las reglas ECA proporcionan a los sistemas de bases de datos la capacidad de reaccionar ante ciertos eventos como inserciones, modificaciones o consultas. Cuando suceden estos eventos, se activan una o varias reglas al mismo tiempo, en este último caso se usan los mecanismos de resolución de conflictos para elegir que regla se debe ejecutar.

Se mostraron algunos ejemplos del trabajo que se ha hecho en el campo de las reglas ECA y las bases de documentos XML. En la sección 2.4.2 se describe un artículo en el que se usan reglas activas en repositorios XML para implementar la tecnología *push*. En la sección 2.4.3 se mostró la sintaxis y semántica de un lenguaje que los autores propusieron para especificar reglas ECA en repositorios XML.

Capítulo 3

Descripción Formal de ADM

En este capítulo introduciremos formalmente la sintaxis y la semántica operacional de ADM con el propósito de formalizar el modelo de programación de las reglas deductivas y activas. Utilizaremos el estilo de la semántica operacional para describir las reglas de inferencia de cada operación básica de ADM. La descripción del lenguaje consiste en la presentación de dos modelos fundamentales: el modelo de reglas deductivas y el modelo de reglas activas. La descripción comienza con algunas definiciones básicas sobre el dominio sintáctico de los objetos del lenguaje como constantes, variables y términos así como de los procedimientos de sustitución y unificación en el contexto de XML. A continuación se describe el modelo de reglas deductivas enunciando tanto el algoritmo del procedimiento de resolución como la correctitud del procedimiento. Aquí se discute brevemente la bien conocida interpretación procedural del principio de resolución. Finalmente, se describe el modelo de reglas activas como sistema de reescritura de multiconjuntos de términos XML. Se presentan y discuten las reglas de inferencia para las operaciones de inserción, eliminación, envío y recepción de (fragmentos de) documentos XML.

3.1. Sintaxis

La sintaxis abstracta de un término XML está dada formalmente por la gramática que aparece en la figura 3.1. Un término XML puede ser una cadena de caracteres, un texto, una variable, un elemento XML o un elemento XML con variables. Una cadena de caracteres es una secuencia de caracteres delimitados por comillas simples o dobles y un texto es una secuencia de caracteres delimitados por elementos XML. Una variable es un identificador que comienza con un signo de pesos. Un término es un elemento XML que puede incluir variables en su lista de atributos o en sus subelementos. El término XML $\langle \mathbf{a} \ a_1 = "A_1" \ \dots \ a_m = "A_m" \rangle \ T_1 \ \dots \ T_n \ \langle / \mathbf{a} \rangle$ está *normalizado* si su lista de pares atributo-valor está ordenada lexicográficamente en orden creciente por el nombre del atributo, es decir, $\mathbf{a}_i \leq \mathbf{a}_j$ si $i \leq j$ para todo $i, j \in \{1, \dots, m\}$.

Las *sustituciones* $\Sigma = \mathbf{XMLVariable} \rightarrow \mathbf{XMLTerm}$ son funciones parciales de variables lógicas a términos XML como se define en la Tabla 3.2. La *composición* de sustituciones σ y σ' se escribe $\sigma\sigma'$. La sustitución nula ϵ es la identidad para la composición de sustituciones $\epsilon\sigma = \sigma\epsilon = \sigma$. Una *sustitución concretizada* (*ground substitution*) es aquella que no introduce alguna variable. La *extensión natural* $\sigma : \mathbf{XMLTerm} \rightarrow \mathbf{XMLTerm}$ de sustituciones de variables a términos XML se denota por el mismo nombre. La *instancia* de un término XML T bajo la sustitución σ se denota por $T\sigma$ y su definición inductiva, dada en la Tabla 3.2, queda determinada por la estructura del término XML. Las instancias de términos XML bajo substitutiones concretizadas se llaman *instancias concretizadas*.

El *unificador* σ es una sustitución que produce instancias sintácticamente idénticas $T\sigma = S\sigma$

$$\begin{array}{ll}
a, b, \dots & \in \text{XMLConstant} \\
\$X, \$Y, \dots & \in \text{XMLVariable} \\
A, B, \dots, S, T, \dots & \in \text{XMLTerm} \\
\sigma, \dots & \in \Sigma = \text{XMLVariable} \rightarrow \text{XMLTerm} \\
P, \dots & \in \text{XMLProgram} = M \text{XMLTerm} \\
\\
T, A ::= & a \\
& | \$X \\
& | \langle a \ a_1 = "A_1" \ \dots \ a_m = "A_m" \ \rangle / \rangle \\
& | \langle a \ a_1 = "A_1" \ \dots \ a_m = "A_m" \ \rangle T_1 \ \dots \ T_n \langle /a \rangle
\end{array}$$

Figura 3.1: Sintaxis abstracta de términos XML.

$$\begin{array}{ll}
a \sigma & = a \\
\$X \sigma & = \sigma(\$X) \\
\langle a \ a_1 = "A_1" \ \dots \ a_m = "A_m" \ \rangle / \rangle \sigma & = \langle a \ a_1 = "A_1" \ \sigma \ \dots \ a_m = "A_m" \ \sigma \ / \rangle \\
\boxed{\begin{array}{l} \langle a \ a_1 = "A_1" \ \dots \ a_m = "A_m" \ \rangle \\ T_1 \ \dots \ T_n \\ \langle /a \rangle \end{array}} \sigma & = \boxed{\begin{array}{l} \langle a \ a_1 = "A_1" \ \sigma \ \dots \ a_m = "A_m" \ \sigma \ \rangle \\ T_1 \sigma \ \dots \ T_n \sigma \\ \langle /a \rangle \end{array}}
\end{array}$$

Figura 3.2: Substitución de variables en términos XML.

de los términos XML T y S bajo σ . El *unificador mas general* (*mgu*) es un unificador que no se puede obtener por la composición de otros distintos. El *algoritmo de unificación*, mostrado en la Tabla 3.3, corresponde esencialmente con aquel introducido por Martelli-Montanari y produce, ya sea, el unificador mas general si existe o una falla si no existe unificador.

El unificador más general se define en la cerradura reflexiva y transitiva de la relación binaria $\triangleright : (\text{XMLTerm}, \Sigma) \rightarrow (\text{XMLTerm}, \Sigma)$. La unificación de términos XML provee un mecanismo uniforme para el paso de parámetros, y para la construcción y selección de la información contenida en el documento XML.

3.2. Modelo de Reglas Deductivas

El modelo de reglas deductivas de ADM coincide con aquel de la programación lógica el cual se basa en los principios de unificación y de resolución SLD. La razón de haber adoptado este modelo es la bien establecida fundamentación teórica que posee así como el amplio conocimiento que se tiene de él. Los enfoques de implementación generalmente coinciden en la bien conocida interpretación procedural sugerida por Kowalski del principio de resolución SLD. Recíprocamente, la lectura declarativa de un procedimiento lógico establece la validez de la llamada del encabezado del procedimiento si todas las condiciones que forman su cuerpo son válidas.

Un *paso de resolución SLD* se define por la relación binaria $\rightarrow : (\text{XMLTerm}, \Sigma) \rightarrow (\text{XMLTerm}, \Sigma)$ que transforma la consulta $(\langle \text{if} \rangle C_1 C_2 \dots C_n \langle / \text{if} \rangle, \sigma)$ en la consulta: $(\langle \text{if} \rangle B_1 \sigma' \dots B_m \sigma' C_2 \sigma' \dots C_n \sigma' \langle / \text{if} \rangle, \sigma \sigma')$, reemplazando la llamada C_1 al procedimiento $\langle a \ a_1 = "A_1" \ \dots \ a_k = "A_k" \ \rangle / \rangle$ por la instancia bajo σ' de su cuerpo $B_1 \sigma' \dots B_m \sigma'$, siempre que exista el unificador mas general σ' de C_1 y $\langle a \ a_1 = "A_1" \ \dots \ a_k = "A_k" \ \rangle / \rangle$.

La *respuesta calculada* de una consulta se obtiene a partir de cero, uno o más pasos de resolución SLD comenzando desde la consulta inicial. La ejecución del programa termina cuando

$$\begin{array}{c}
\begin{array}{|l}
\langle a \ a_1 = "A_1" \ \dots \ a_m = "A_m" \rangle \\
S_1 \dots S_n \\
\langle /a \rangle
\end{array} \\
S = \\
\hline
(C \cup \{S = T\}, \sigma) \triangleright (C \cup \{A_1 = B_1, \dots, A_m = B_m, S_1 = T_1, \dots, S_n = T_n\}, \sigma) \\
\\
\begin{array}{|l}
\langle a \ a_1 = "B_1" \ \dots \ a_m = "B_m" \rangle \\
T_1 \dots T_n \\
\langle /a \rangle
\end{array} \\
T = \\
\hline
(C \cup \{S = T\}, \sigma) \triangleright (C \cup \{A_1 = B_1, \dots, A_m = B_m\}, \sigma) \\
\\
\frac{S = \langle a \ a_1 = "A_1" \ \dots \ a_m = "A_m" \ / \rangle \quad T = \langle a \ a_1 = "B_1" \ \dots \ a_m = "B_m" \ / \rangle}{(C \cup \{S = T\}, \sigma) \triangleright (C \cup \{A_1 = B_1, \dots, A_m = B_m\}, \sigma)} \\
\\
(C \cup \{T = T\}, \sigma) \triangleright (C, \sigma) \\
(C \cup \{T = \$X\}, \sigma) \triangleright (C \cup \{\$X = T\}, \sigma) \\
(C \cup \{\$X = T\}, \sigma) \triangleright (C \{\$X \mapsto T\}, \sigma \{\$X \mapsto T\}) \quad \text{si } \$X \notin \text{vars}(T) \\
(C \cup \{T = T\}, \sigma) \triangleright \text{failure} \quad \text{en otro caso} \\
\\
\frac{(\{T = S\}, \epsilon) \triangleright^* (\epsilon, \sigma)}{\text{mgu}(T, S) = \sigma}
\end{array}$$

Figura 3.3: Unificación de términos XML.

$$\frac{\begin{array}{l} \langle a \ a_1 = "A_1" \ \dots \ a_k = "A_k" \rangle \ B_1 \ \dots \ B_m \langle /a \rangle \in \text{XMLProgram} \\ \exists \sigma'. \ \sigma' = \text{mgu}(\text{head}(C_1), \langle a \ a_1 = "A_1" \ \dots \ a_k = "A_k" \rangle) \end{array}}{\langle \text{if} \rangle C_1 C_2 \dots C_n \langle / \text{if} \rangle, \sigma \rightarrow \langle \text{if} \rangle B_1 \sigma' \dots B_m \sigma' C_2 \sigma' \dots C_n \sigma' \langle / \text{if} \rangle, \sigma \sigma'}$$

Figura 3.4: Relación de inferencia SLD.

$$\frac{(\langle \text{if} \rangle C_1 \dots C_n \langle / \text{if} \rangle, \epsilon) :^* (\langle \text{if} \rangle \langle / \text{if} \rangle, \sigma)}{P \models_{\sigma} \langle \text{if} \rangle C_1 \dots C_n \langle / \text{if} \rangle}$$

Figura 3.5: Correctitud de la relación de resolución SLD.

ya no hay más invocaciones de procedimientos que resolver en la consulta. La respuesta calculada se obtiene de la composición de las sustituciones usadas en cada paso del procedimiento de resolución. La figura 3.5 establece que la respuesta calculada es efectivamente la respuesta correcta, es decir, la sustitución σ es la solución de la consulta. Un resultado bien conocido de la programación lógica dice que una respuesta calculada σ obtenida por el cálculo de resolución SLD es un modelo para ambos el programa P y la consulta $\langle \text{if} \rangle C_1 \cdots C_n \langle \text{/if} \rangle$. Por lo tanto, las respuestas calculadas son aquellas que satisfacen las condiciones lógicas del programa.

3.3. Modelo de Reglas Activas

La definición del comportamiento reactivo y la preservación de las restricciones de integridad se describen mediante *reglas activas*, también conocidas como *reglas evento-condición-acción (ECA)*. El detector de eventos percibe y notifica al sistema cuando una colección de documentos XML ha cambiado debido a que un nuevo documento fue insertado, uno ya existente fue eliminado o algún otro fue recibido. Formalmente, la colección de documentos XML se representa por un multiconjunto de términos XML cuyo tipo se denota por **MXMLTerm**. La representación de una colección de documentos XML como multiconjunto obedece al hecho de que la multiplicidad de los documentos es importante aún cuando las relaciones espaciales relativas que puedan guardar los documentos entre sí no lo sean. En lo sucesivo designaremos como *programa* a la colección $P \in \mathbf{MXMLTerm}$ de documentos formada por las bases extensionales e intensionales de documentos XML. Esta designación se justifica por el hecho de que la colección entera de documentos incluye a los documentos XML ordinarios (bases de datos extensionales) y a las reglas deductivas y activas (bases de datos intensionales).

3.3.1. Ambito de una regla

Aunque el modelo subyacente de repositorio compartido de documentos es un principio fundamental de diseño en ADM, la partición de la colección global de documentos simplifica el modelo de programación porque permite identificar colecciones individuales. Por esta razón, la colección global de documentos se encuentra particionada en localidades referidas por un nombre simbólico. Cada colección puede a su vez particionarse de nuevo para dar lugar así a una estructura jerárquica similar a la estructura de árbol de un directorio de archivos. Esta organización permite localizar con facilidad a cualquier documento dentro de la colección global.

Una *localidad* define una partición jerárquica en la colección de documentos la cual puede designarse por un nombre simbólico. La notación P/L designa a la colección de documentos ubicados en la localidad L del programa o colección P , en tanto que la notación $P/L/D$ designa a un documento D ubicado en la localidad L de P . El ámbito de una regla activa se extiende a la localidad en donde se encuentra el documento que contiene a la regla. La importancia de la noción de ámbito radica en que restringe los documentos considerados durante el proceso de consulta usado en el modelo deductivo. De esta forma, la designación de documentos por localidades restringe en forma controlada la selección y activación de reglas activas. En lo sucesivo, por colección nos referimos indistintamente a la colección global P o la subcolección P/L de P en la localidad L .

3.3.2. Selección

Al procedimiento que permite determinar las reglas que pueden ser ejecutadas como respuesta a la ocurrencia de un evento externo se le llama *selección*. Una vez que el administrador de eventos ha indicado la ocurrencia del evento A_e , la estructura del documento asociado con el

$$\begin{array}{c}
 R = \boxed{\begin{array}{l}
 \langle \text{rule} \rangle \\
 \langle \text{on} \rangle E \langle / \text{on} \rangle \\
 \langle \text{if} \rangle C \langle / \text{if} \rangle \\
 \langle \text{do} \rangle A \langle / \text{do} \rangle \\
 \langle / \text{rule} \rangle
 \end{array}} \\
 \\
 \frac{P \cup \{A_e\} \models_{\sigma} \langle \text{if} \rangle E C \langle / \text{if} \rangle}{P, A_e \models_{\sigma} R} \\
 \\
 \frac{\neg \exists \sigma \in \Sigma. P \cup \{A_e\} \models_{\sigma} \langle \text{if} \rangle E C \langle / \text{if} \rangle}{P, A_e \not\models R}
 \end{array}$$

Figura 3.6: Selección de una regla por la ocurrencia de un evento.

evento determina la selección de la regla R cuando el elemento evento E de R posee la misma estructura que A_e . El procedimiento de unificación puede utilizarse para determinar la igualdad de las estructuras y al mismo tiempo recuperar la información σ asociada con el evento A_e de tal manera que $E\sigma = A_e\sigma$. La estructura e información que se obtienen a partir del contenido del documento XML participante en el evento permite seleccionar todas las reglas que atienden exactamente al mismo tipo de evento. Este grado de indeterminismo es aceptable cuando se busca garantizar la imparcialidad en la selección de las reglas que rigen el comportamiento de los sistemas distribuidos. De las reglas seleccionadas, se escogen solamente aquellas cuya condición es satisfecha por la información recuperada del evento. Utilizando el procedimiento de resolución SLD, si la condición $C\sigma$ se cumple (por refutación), se ejecutan entonces las acciones $A\sigma$ de la regla, que pueden incluir operaciones de inserción, eliminación y recepción de documentos de la colección.

La figura 3.6 formaliza la selección y activación de una regla mediante la relación de *selección* \models de una regla. La relación de selección $P, A_e \models_{\sigma} R$ se cumple siempre que ambos el evento E y la condición C de la regla R sean consecuencia lógica del contexto formado por las bases extensionales de P (temporalmente) con el evento A_e . En tal caso, existe una respuesta calculada σ que es la solución de la consulta extendida $\langle \text{if} \rangle E C \langle / \text{if} \rangle$ en la colección de documentos $P \oplus \{A_e\}$. Como se indica en la inclusión del evento E en la consulta, la estructura del documento A_e debe coincidir con la estructura dada por E .

La segunda regla de inferencia establece que la relación $P, A_e \not\models R$ de activación no se cumple en la colección de documentos cuando no existe una respuesta calculada que satisfaga a la condición y al evento de la regla. La relación de activación de una regla es un concepto fundamental ya que determina cuando una regla se puede ejecutar o no. Esta relación la usaremos en todas las reglas de inferencia de las operaciones sobre documentos que se describen en adelante.

3.3.3. Activación

La semántica operacional de ADM se define mediante las nociones de configuración y de transición. Una *configuración no terminal* $(P, \sigma) \in \mathbf{MXMLTerm} \times \Sigma$ consiste de un programa P (base intensional y extensional de documentos XML) y de un asignamiento (substitución) Σ de términos XML a variables. Una *configuración terminal* P consiste simplemente un programa $P \in \mathbf{MXMLTerm}$. La relación de *transición* entre configuraciones $\longrightarrow: (\mathbf{MXMLTerm} \times \Sigma) \rightarrow (\mathbf{MXMLTerm} \times \Sigma \cup \mathbf{MXMLTerm})$ establece la evolución del comportamiento del sistema por las interacciones de sus participantes. La relación de transición es *terminante* cuando cada secuencia de transiciones $(P_0, \sigma_0) \longrightarrow (P_1, \sigma_1) \longrightarrow \dots \longrightarrow (P_{n-1}, \sigma_{n-1}) \longrightarrow P_n$ es finita y resulta en una

$$\begin{array}{c}
 R = \boxed{\begin{array}{l}
 \langle \text{rule} \rangle \\
 \langle \text{on} \rangle E \langle / \text{on} \rangle \\
 \langle \text{if} \rangle C \langle / \text{if} \rangle \\
 \langle \text{do} \rangle A \langle / \text{do} \rangle \\
 \langle / \text{rule} \rangle
 \end{array}} \\
 \\
 \frac{\exists R \in P. P, A_e \models_{\sigma'} R}{(P \oplus \{A_e\}, \sigma) \longrightarrow (P \oplus \{A\}, \sigma\sigma')} \\
 \\
 \frac{\forall R \in P. P \not\models R}{(P, \sigma) \longrightarrow P\sigma}
 \end{array}$$

Figura 3.7: Reglas de inferencia para la activación de reglas.

configuración terminante. Por el contrario, la transición es *divergente* cuando es no terminante.

En la figura 3.7 aparecen las reglas de inferencia para la relación de activación la cual incluye a su cerradura reflexiva transitiva. Asumiendo que la regla R en P se compone del evento E , la condición C y la acción A , la primera regla de inferencia establece que la ocurrencia del evento A_e causa la substitución de $P \oplus \{A_e\}$ (la colección P extendida con el documento asociado al evento A_e) por $P\sigma' \oplus \{A\sigma'\}$ (la instancia de la colección $P\sigma'$ extendida con las instancias de las acciones $A\sigma'$ de R) bajo σ' (la respuesta calculada obtenida al demostrar que R es seleccionada por A_e en P).

La ocurrencia explícita de la instancia $P\sigma'$ de la colección de documentos en la relación de transición establece la propagación de los fragmentos de documentos XML vinculados a las variables en todos los documentos donde ellas ocurran. Mediante este mecanismo de propagación, el contenido de los documentos XML se pueden ir construyendo a medida que las variables se van concretizando como resultado de la ejecución de las reglas activas.

Puesto que pueden ser varias las reglas seleccionadas para la creación de instancias, la selección final de la instancia se determina por la aplicación de las reglas de inferencia de interacción entre reglas activas dadas más adelante (figura 3.8).

La segunda regla de inferencia establece la condición de terminación de la transición cuando ninguna regla puede activarse por la ocurrencia de A_e en P . En este caso, la secuencia de transiciones siempre resulta en una configuración terminante, dejando como resultado a la instancia $P\sigma$ de la colección que se obtiene al substituir todos los fragmentos vinculados a las variables en aquellos documentos donde estas aparecen.

3.3.4. Interacción entre Reglas Activas

Los programas o colecciones de documentos pueden interactuar entre ellos dependiendo de los documentos que compartan. En la figura 3.8 se muestran las reglas de inferencia que describen el comportamiento global de dos programas que interactúan. La función *docset* de la regla R en el programa P determina el multiconjunto de todos los documentos que hacen a R seleccionable para su activación. El propósito de la función *docset* es determinar aquellos documentos que pueden hacer elegibles a más de una regla, lo que señalaría a dicho multiconjunto como una región crítica de documentos compartidos por reglas. Las reglas de inferencia de interacción entre programas deben, por lo tanto, garantizar el acceso exclusivo de una regla a la vez de entre varias seleccionadas.

Las primeras dos reglas se aplican cuando ambos programas comparten un multiconjunto no vacío de documentos. En tal caso cualquiera de los dos programas, pero no ambos simultáneamen-

$$\begin{array}{c}
\frac{(P_1, \sigma) \longrightarrow (P'_1, \sigma') \quad docset_P(P_1\sigma) \cap docset_P(P_2\sigma) \neq \emptyset}{(P_1 \oplus P_2, \sigma) \longrightarrow (P'_1\sigma' \oplus P_2, \sigma\sigma')} \quad \frac{(P_2, \sigma) \longrightarrow (P'_2, \sigma') \quad docset_P(P_1\sigma) \cap docset_P(P_2\sigma) \neq \emptyset}{(P_1 \oplus P_2, \sigma) \longrightarrow (P_1 \oplus P'_2\sigma', \sigma\sigma')} \\
\\
\frac{(P_1, \sigma) \longrightarrow (P'_1, \sigma_1) \quad (P_2, \sigma) \longrightarrow (P'_2, \sigma_2) \quad docset_P(P_1\sigma) \cap docset_P(P_2\sigma) = \emptyset}{(P_1 \oplus P_2, \sigma) \longrightarrow (P'_1\sigma_1 \oplus P'_2\sigma_2, \sigma\sigma_1\sigma_2)} \\
\\
\frac{(P_1, \sigma) \longrightarrow P_1\sigma \quad (P_2, \sigma) \longrightarrow P_2\sigma}{(P_1 \oplus P_2, \sigma) \longrightarrow P_1\sigma \oplus P_2\sigma} \\
\\
docset_P(R) = \{doc(A_e) \mid P, A_e \models R\} \\
\\
docset_P(\uplus R) = \uplus docset_P(R)
\end{array}$$

Figura 3.8: Reglas de inferencia para la interacción de programas.

$$\begin{array}{c}
\frac{(P \oplus \{P_1\}, \sigma) \longrightarrow P\sigma \oplus \{P_1\}\sigma}{(P \oplus \{\langle seq \rangle P_1 P_2 \langle /seq \rangle\}, \sigma) \longrightarrow (P\sigma \oplus \{P_1\}\sigma \oplus \{P_2\}, \sigma)} \\
\\
\frac{(P \oplus \{P_1\}, \sigma) \longrightarrow (P \oplus \{P'_1\}, \sigma')}{(P \oplus \{\langle seq \rangle P_1 P_2 \langle /seq \rangle\}, \sigma) \longrightarrow (P \oplus \{\langle seq \rangle P'_1 P_2 \langle /seq \rangle\}, \sigma\sigma')} \\
\\
(P \oplus \{\langle par \rangle P_1 P_2 \langle /par \rangle\}, \sigma) \longrightarrow (P \oplus \{P_1\} \oplus \{P_2\}, \sigma)
\end{array}$$

Figura 3.9: Reglas de inferencia para la composición secuencial y paralela de programas.

te, puede desarrollar su comportamiento. La tercera regla se aplica solamente si los programas no comparten alguna colección de documentos. En tal caso, si ambos programas exhiben algún progreso por separado, entonces los programas no interfieren entre si y podrán desarrollar su comportamiento independientemente el uno del otro. Finalmente, la última regla describe la condición de terminación. En este caso, si dos programas terminan independientemente, también lo hacen uniendo las colecciones de documentos de ambos programas.

3.3.5. Secuencialidad y Concurrencia

La ejecución de las reglas puede dar lugar a comportamientos inaceptables debido a que el orden en la ejecución de las reglas puede causar interferencia entre ellas impidiendo la selección de unas o promoviendo la ejecución reiterada de otras. Para reducir el no determinismo en el orden en el que las reglas se ejecutan, se introduce la composición secuencial y concurrente para restringir el orden de su ejecución.

La figura 3.9 muestra las reglas de inferencia para la composición secuencial y concurrente de programas, las cuales se introducen mediante las marcas $\langle seq \rangle$ y $\langle par \rangle$, respectivamente. La primera regla de inferencia describe la condición de terminación de dos programas bajo la composición secuencial. Si el programa P_1 termina en el estado σ , entonces la composición secuencial de las reglas P_1 y P_2 se comporta como P_2 en el mismo estado σ . En este caso, puesto que P_1 es terminante no existe interacción entre los subprogramas P y $\{P_1\}$ en el estado σ , el programa $P \oplus \{P_1\} \oplus \{P_2\}$ la interacción tendrá lugar solamente entre $P \oplus \{P_1\}$ y $\{P_2\}$.

La segunda regla de inferencia describe la condición de progreso de dos reglas activas bajo la composición secuencial. Si en el estado σ la regla P_1 reduce a la regla P'_1 , entonces la composición secuencial de P_1 y P_2 reduce a la composición secuencial de P'_1 y P_2 . Por reducción de la regla P_1 a la regla P'_1 nos referimos a la transición entre configuraciones que convierte una regla activa R a su acción siempre que exista un evento A_e que active a R .

La capacidad de describir secuencialidad y concurrencia en las acciones que desarrollan los programas tiene consecuencia inmediata en los modelos de comunicación disponibles. En ADM existen dos modelos principales de sincronización y de comunicación entre programas conocidos como modelo de espacio compartido de documentos y modelo de intercambio síncrono de mensajes. En la sección 3.3.6 trataremos el modelo de espacio compartido, dejando para la sección 3.3.7 la presentación del modelo de intercambio de mensajes.

3.3.6. Comunicación Asíncrona

En el *modelo de espacio compartido de documentos*, los programas interactúan mediante las operaciones de inserción y eliminación de documentos que realizan sobre el espacio de documentos. El espacio puede estar particionado en localidades por lo que los programas pueden interactuar siempre que indiquen explícitamente la localidad de los documentos a que hacen referencia.

La figura 3.10 muestra las reglas de inferencia para la inserción de documentos. La primera regla de inferencia describe el efecto de activar la regla R después de detectar un evento de inserción, lo que resulta en modificar la colección P/L realizando las acciones $A\sigma'$ de R e insertando al documento D_e . La segunda regla de inferencia describe el comportamiento que tiene una acción de inserción cuando no existe una regla activa en la colección local P/L que perciba al evento. En tal caso, el documento D_e se inserta simplemente en la colección. La tercera regla de inferencia describe como la acción `<insert into=L>D_e</insert>` con origen en la colección P/M se traduce en la acción `<insert from=M>D_e</insert>` con ámbito en la colección P/L . Intuitivamente, esta regla de inferencia causa que la acción de inserción sea transferida hacia la colección en donde tendrá efecto local, al mismo tiempo que modifica su estructura para revelar su origen.

Es importante resaltar el hecho que el evento generado por la acción `<insert into=L>D_e</insert>` es percibido únicamente por las reglas de P/L , mientras que para las reglas de otras colecciones dicho evento permanece imperceptible.

La presencia del atributo `from` en el evento determina si la regla puede seleccionarse dependiendo de la localidad de la colección donde se originó la inserción. Sin embargo, cuando no se especifica el atributo `from`, entonces la regla puede seleccionarse independientemente del origen de la inserción.

Las reglas de inferencia para la eliminación de documentos se muestran en la figura 3.11. La primera regla de inferencia describe el comportamiento de R cuando se detecta un evento de eliminación, lo que resulta en modificar la colección P/L , eliminando el documento D_e que hace elegible a R y realizando las acciones $A\sigma'$ de R . La segunda regla de inferencia describe el efecto que desarrolla una acción de eliminación cuando no existe una regla en la colección P/L que sea seleccionable. En consecuencia, la única acción realizada es que el documento D_e se elimina de la colección. La tercera regla de inferencia traduce la acción `<delete from=L>D_e</delete>` con origen en la colección P/M en la acción `<delete by=M>D_e</delete>` con destino en la colección P/L en donde se ejecuta R . Al igual que en la operación de inserción, esta regla transfiere la acción de eliminación desde cualquier colección en P a la colección destino donde tendrá finalmente efecto.

Las operaciones de inserción y eliminación de documentos en el espacio compartido constituyen un modelo de comunicación desacoplado asíncrono (no bloqueante) que es de gran utilidad en

$$\begin{array}{c}
R = \boxed{\begin{array}{l}
\langle \text{rule} \rangle \\
\langle \text{on} \rangle \langle \text{insert from} = M \rangle D \langle \text{/insert} \rangle \langle \text{/on} \rangle \\
\langle \text{if} \rangle C \langle \text{/if} \rangle \\
\langle \text{do} \rangle A \langle \text{/do} \rangle \\
\langle \text{/rule} \rangle
\end{array}} \\
\\
\frac{\exists R \in P/L. \exists \sigma' \in \Sigma. P/L, D_e \models_{\sigma'} R}{(P/L \oplus \{ \langle \text{insert from} = M \rangle D_e \langle \text{/insert} \rangle \}, \sigma)} \\
\longrightarrow (P/L \oplus \{ A \sigma', D_e \}, \sigma \sigma') \\
\\
\frac{\forall R \in P/L. P/L, D_e \not\models R}{(P/L \oplus \{ \langle \text{insert from} = M \rangle D_e \langle \text{/insert} \rangle \}, \sigma)} \\
\longrightarrow P/L \oplus \{ D_e \} \\
\\
(\{ P/M / \langle \text{insert into} = L \rangle D_e \langle \text{/insert} \rangle \}, \sigma) \\
\longrightarrow (\{ P/L / \langle \text{insert from} = M \rangle D_e \langle \text{/insert} \rangle \}, \sigma)
\end{array}$$

Figura 3.10: Reglas de inferencia para la inserción de documentos.

varios dominios de aplicación como la gestión documental. Sin embargo, la naturaleza asíncrona de este modelo dificulta la coordinación de programas que deben esperar por información crítica hasta que esta sea disponible. Dicha dificultad radica en la presencia de reglas de inferencia que se aplican en aquellos casos en donde no existen reglas activas seleccionables. Aunque en principio, al eliminar estas reglas de inferencia se obtendría un modelo de comunicación bloqueante, el modelo vería comprometida su completitud (existencia de transiciones terminantes), un tema que no se discutirá en esta tesis pero que sin duda es de gran interés teórico en la fundamentación de ADM. Para resolver el problema de contar con un medio de comunicación síncrono, en la siguiente sección presentaremos el modelo de comunicación erigido sobre el intercambio de mensajes.

3.3.7. Comunicación Síncrona

El *modelo de intercambio de mensajes* introduce un modelo de comunicación síncrona que extiende el modelo de programación de ADM. Este modelo de comunicación permite suspender el progreso en la ejecución de un programa mientras no se detecte un evento de recepción de un documento con la estructura y el contenido adecuados.

Las reglas de inferencia que se presentan en la figura 3.12 describen la recepción y el envío de documentos XML. La primera regla de inferencia describe el efecto que se produce al detectar un evento de recepción en la colección P/L en donde existe una regla R que responde al evento. Como resultado, la colección P/L se modifica insertando el documento recibido D_e y ejecutando las acciones $A\sigma'$ de R . La segunda regla describe el efecto de la acción localizada en la colección P/M de enviar el documento D_e a la colección P/L .

A diferencia del modelo de espacio de documentos, en este modelo no existe una regla complementaria de inferencia para el caso en el que no existe una regla activa que responda al evento de recepción. En consecuencia, para aquellos mensajes que no habilitan regla alguna, no se produce ningún cambio en el estado del programa. Si por otra parte existiera la regla complementaria de inferencia, entonces el programa terminaría aunque sin lograr comunicación alguna.

$$\begin{array}{c}
R = \boxed{\begin{array}{l}
\langle \text{rule} \rangle \\
\langle \text{on} \rangle \langle \text{delete by} = M \rangle D \langle / \text{delete} \rangle \langle / \text{on} \rangle \\
\langle \text{if} \rangle C \langle / \text{if} \rangle \\
\langle \text{do} \rangle A \langle / \text{do} \rangle \\
\langle / \text{rule} \rangle
\end{array}} \\
\\
\frac{\exists R \in P/L. \exists \sigma' \in \Sigma. P/L, D_e \models_{\sigma'} R}{(P/L \oplus \{D_e, \langle \text{delete by} = M \rangle D_e \langle / \text{delete} \rangle\}, \sigma) \longrightarrow (P/L \oplus \{A \sigma'\}, \sigma \sigma')} \\
\\
\frac{\forall R \in P/L. P/L, D_e \not\models R}{(P/L \oplus \{D_e, P/L / \langle \text{delete by} = M \rangle D_e \langle / \text{delete} \rangle\}, \sigma) \longrightarrow P/L} \\
\\
(\{P/M / \langle \text{delete from} = L \rangle D_e \langle / \text{delete} \rangle\}, \sigma) \longrightarrow (\{P/L / \langle \text{delete by} = M \rangle D_e \langle / \text{delete} \rangle\}, \sigma)
\end{array}$$

Figura 3.11: Relación de reducción para la eliminación de documentos.

$$\begin{array}{c}
R = \boxed{\begin{array}{l}
\langle \text{rule} \rangle \\
\langle \text{on} \rangle \langle \text{receive from} = M \rangle D \langle / \text{receive} \rangle \langle / \text{on} \rangle \\
\langle \text{if} \rangle C \langle / \text{if} \rangle \\
\langle \text{do} \rangle A \langle / \text{do} \rangle \\
\langle / \text{rule} \rangle
\end{array}} \\
\\
\frac{\exists R \in P/L. \exists \sigma' \in \Sigma. P/L, D_e \models_{\sigma'} R}{(P/L \oplus \{\langle \text{receive from} = M \rangle D_e \langle / \text{receive} \rangle\}, \sigma) \longrightarrow (P/L \oplus \{A \sigma', D_e\}, \sigma \sigma')} \\
\\
(\{P/M / \langle \text{send to} = L \rangle D_e \langle / \text{send} \rangle\}, \sigma) \longrightarrow (\{P/L / \langle \text{receive from} = M \rangle D_e \langle / \text{receive} \rangle\}, \sigma)
\end{array}$$

Figura 3.12: Reglas de inferencia para la recepción y el envío de documentos.

El propósito de la segunda regla de inferencia es describir la semántica de la operación de envío, interpretada en términos de su operación complementaria, como la recepción del documento en el destino del envío. Como puede apreciarse, la segunda regla de inferencia no hace alusión a la infraestructura tecnológica del protocolo de transporte usado en la comunicación. Este es el tema del capítulo 6 en donde se discute la implementación de ADM. En el capítulo siguiente, se estudiará un caso de estudio que mostrará que las construcciones del lenguaje y del modelo de programación de ADM son suficientemente expresivos para modelar esquemas de interacción complejos como los sistemas de cache de páginas Web. De nuevo, en el capítulo 6 se presentará la implementación de este caso de estudio.

3.4. Resumen

En este capítulo hemos presentado la descripción de la semántica formal del modelo de programación de ADM. El modelo de programación puede clasificarse como un sistema de reescritura de multiconjuntos de términos (XML). Las reglas de reescritura, conocidas en ADM como reglas activas, describen interacciones entre colecciones de programas mediante transformaciones de un espacio compartido, jerárquicamente particionado, de documentos XML. Las transformaciones ocurren por la aplicación de operaciones de inserción, eliminación, envío y recepción de (fragmentos de) documentos siempre que el contenido de los documentos cumpla con condiciones bien definidas. La definición y verificación de dichas condiciones se establece como reglas deductivas cuyo modelo de programación tiene fundamento en los principios de unificación y de resolución de la programación lógica. Una demostración informal de la expresividad de ADM se presentará en el siguiente capítulo mediante la descripción de un sistema de cache de páginas Web así como de su desarrollo de acuerdo al comportamiento previsto en las reglas. La factibilidad de la implementación de ADM quedará establecida en un capítulo posterior cuando se presente la realización de las primitivas propuestas que permiten aprovechar la infraestructura tecnológica disponible de las bases de datos y de los servicios Web.

Capítulo 4

Caso de estudio: Proxy http

4.1. Introducción

Con el propósito de mostrar las nuevas capacidades del lenguaje ADM, se propone el caso de estudio de un proxy http. En el apéndice ?? se dan más detalles del funcionamiento de estas aplicaciones.

Dado que la mayoría de los documentos que se solicitan en la Web son documentos estáticos (páginas de inicio o *home pages*, archivos de audio y de vídeo etc), se ha considerado la posibilidad de almacenar tales objetos como una forma de reducir el tráfico en la Web. Una forma común hacer esto son los servidores proxy para páginas Web, conocidos como proxy http, proxy Web o proxy-caché. Los cuales son intermediarios entre procesos de navegador y servidores de Web sobre la Internet [C197]. Algunas de las ventajas al usar servidores proxy http son:

- Reducen la carga sobre servidores Web ocupados.
- Reducen la latencia para obtener datos Web porque los clientes pueden obtener los datos de un proxy http local en vez de pedir los datos directamente del sitio que los proporciona [C97].
- Permiten que muchos usuarios accedan a Internet mediante una sola conexión.
- Numerosos estudios han mostrado que el porcentaje de accesos exitosos a la caché del proxy (conocidos como *hit ratio*) puede ser el 50% o mas. Esto quiere decir que si los proxy caché son utilizados extensivamente, pueden reducir el tráfico de red considerablemente [C197].
- Para las empresas y proveedores de contenidos de Internet las ventajas se concretan en la posibilidad de atender un número mayor de visitas a sus páginas Web sin necesidad de hacer inversiones adicionales, minimizando el riesgo de colapso y saturación.

4.2. Descripción del proxy http en ADM

En nuestro sistema proxy http hay cuatro participantes principales que son: el cliente, el proxy http, el servidor interno y el servidor externo. El cliente puede ser un navegador o un programa que hace una petición de una página Web vía Internet a un servidor externo. Para que la petición del cliente llegue al servidor externo debe pasar primero por dos servidores intermedios que son el proxy http y el servidor interno.

El proxy http es un servidor intermedio entre el cliente y el servidor externo que se encarga de hacer peticiones de páginas (cuando es la primera vez que se solicitan) y almacenar las páginas

(cuando ya se pidieron al menos una vez). El Proxy http hace las solicitudes construyendo una petición de un servicio Web y enviándola al servidor externo.

El servidor interno es propiamente dicho la caché de nuestro proxy, pues en este servidor se encuentran almacenadas las páginas que se han pedido alguna vez y los datos de estas páginas. Para almacenar la información, el servidor interno requiere usar dos BD. Una BD contiene todas las páginas Web que se han pedido al menos una vez, esta base de documentos la llamamos *Registry*. La otra BD es la que tiene un registrada cada una de las páginas que se encuentran almacenadas la llamamos *Directory*.

El servidor externo es un servidor común de páginas Web. Sin embargo por simplicidad en esta tesis se implementó como un servicio Web en lugar de usar el protocolo HTTP. Este criterio de diseño no impide que este trabajo pueda utilizar el protocolo HTTP (haciendo las modificaciones apropiadas).

En nuestra implementación, el servidor externo trabaja como un intermediario que traduce las peticiones formuladas como servicios Web a peticiones que usan el protocolo HTTP. Proporciona respuesta a peticiones que recibe del proxy http. Los servicios que ofrece son en general la extracción de información de páginas Web. Por ejemplo de una página Web específica, a solicitud, puede devolverse sólo el encabezado de la misma. Por esto, el servidor externo consulta al servidor Web donde se encuentra la página y una vez que la obtiene, extrae la información solicitada por el proxy y la envía al mismo. Las consultas del servidor externo a otros servidores Web se utiliza un mecanismo de comunicación muy sencillo basado en el protocolo http. En adelante se asume que esta comunicación tenga lugar cuando se solicita un servicio al servidor externo y no se mencionará.

Dado que servidor externo usa el protocolo SOAP (recordemos que está implementado como un servicio Web) para el intercambio de mensajes, estos se escriben siguiendo las normas de dicho protocolo. Lo anterior implica que para obtener la información de los mensajes (tanto de peticiones como respuestas), se debe analizar el mensaje SOAP y extraer la información que se requiere. Una petición a un servicio Web, además de estar escrita en formato SOAP debe contener el nombre del servicio que se solicita y los parámetros que se especifican en la definición del servicio (tal especificación se encuentra en el WDSL del servicio). Cuando el Servidor de Servicios Web recibe una petición, extrae la información que requiere (el nombre del servicio y los parámetros) y ejecuta el servicio. En nuestro caso de estudio ocupamos únicamente dos servicios Web cuyos nombres son `InternalServer` y `ExternalServer`. El servicio `ExternalServer` implementa las operaciones `head` y `get` del protocolo HTTP.

La operación `getDate` devuelve la fecha de la modificación más reciente de una página Web. La operación `getDocument` tiene como propósito recuperar el contenido de una página Web.

4.3. Reglas ECA del proxy http

El comportamiento del sistema proxy caché que desarrollamos se modela con reglas ECA, cada uno de los cuatro participantes debe tener sus propias reglas ante las cuales reaccionará cuando sean activadas. En las siguientes subsecciones se muestran las reglas del proxy escritas en XML, escritas en ADM, además se da una descripción de la regla y se muestra el diagrama de secuencia que le corresponde. Las reglas del cliente (reglas 1 y 2) y del servidor externo (reglas 12 y 13) se muestran en el apéndice ??.

En lo sucesivo, cuando se habla de la parte de evento de una regla nos referimos al contenido dentro de las etiquetas `<on>` `</on>`, cuando hablamos de la parte de la condición nos referimos a la parte contenida en las etiquetas `<if>` `</if>` y la parte de la acción de la regla es lo que está dentro de las etiquetas `<do>` `</do>`. Al implementar las reglas en ADM, se hizo necesario

Clase Proxy	
Responsabilidades	Colaboraciones
<ol style="list-style-type: none"> 1. Recibir las solicitudes de páginas y solicitar la fecha de la última actualización de la página solicitada a los servidores externo e interno. 2. Recibir las fechas de la última modificación y determinar de donde se recupera el documento. 3. Recibir las páginas que envía en servidor interno y enviarlas al cliente. 4. Recibir las páginas que envía en servidor externo, almacenarlas en el servidor interno y enviarlas al cliente. 	<ol style="list-style-type: none"> 1. SendingProxy 2. InternalServer 3. ExternalServerBindingImpl

Tabla 4.1: Tarjeta CRC de la clase Proxy.

crear objetos con los nombres de cada una de las etiquetas que se usan para enviar o recibir datos.

4.3.1. Reglas del proxy

El comportamiento del proxy http se define de forma sencilla con 5 reglas únicamente, dichas reglas se describen en las subsecciones siguientes. El proxy utiliza un mecanismo de candado para asegurar la exclusión mutua en la atención de las solicitudes de los clientes, de tal manera que se eviten los problemas de acceso concurrente. La implementación de este mecanismo se explica mas adelante en donde se describe la regla `ProxyAvailable`.

La tarjeta CRC correspondiente a la clase `Proxy` se puede ver en la Tabla 4.1.

Regla 3: PageReqRecByProxy

La regla descrita en la Tabla 4.2 muestra el comportamiento que tiene el proxy http cuando recibe la petición de una página Web. El evento que activa la regla es recibir (etiqueta `<received>`) la petición de un documento, tal petición se indica con el elemento `<getDocument>`, el elemento `<name>` indica el URL de la página que se pide y está almacenado en la variable `$Name`.

La condición que se verifica para ejecutar la acción es: que se encuentre disponible la llave del candado en el proxy (`<lock/>`). En caso de que la llave no se encuentre disponible (por que otro cliente la retiró antes), entonces la regla no se puede ejecutar y en consecuencia el cliente se ve forzado a esperar.

La parte de la acción de la regla es una secuencia de tres acciones: la primera es enviar al servidor interno una petición de la fecha de actualización más reciente del documento pedido (`<getDate>`), la segunda acción es también enviar al servidor externo la petición de la fecha de actualización mas reciente del documento pedido (`<getDate>`). Finalmente la tercera acción es retirar la llave del candado (etiqueta `<delete>`) del proxy (`<lock/>`) para evitar que el proxy atienda otros clientes al mismo tiempo. La Figura 4.1 muestra el diagrama de secuencia de esta regla. La tabla 4.3 tiene el código en ADM de esta regla.

```

<rule name = "PageReqRecByProxy">
  <on>
    <received from = "Client">
      <getDocument><name>$Name</name></getDocument>
    </received>
  </on>
  <if>
    </inserted><lock/></inserted>
  </if>
  <do><sec>
    <delete><key/></delete>
    <send to = "InternalServer">
      <getDate><name>$Name</name></getDate>
    </send>
    <send to = "ExternalServer">
      <getDate><name>$Name</name></getDate>
    </send>
  </sec></do>
</rule>

```

Tabla 4.2: Regla PageReqRecByProxy.

```

public void run()
{
  for (;;)
  {
    GetDocument getDocument = new GetDocument();
    if(cClient.received(getDocument) != null)
    {
      String name = getDocument.name;
      if(cLock.isInserted(key))
      {
        cLock.delete(key);
        GetDate getDate = new GetDate();
        getDate.name = name;
        String uuid = UUID.creaUUID();
        cInternalServer.send(uuid,getDate);
        cExternalServer.send(uuid,getDate);
        cClient.accept();
      }
      else
      {
        cClient.reject();
      }
    }
  }
}

```

Tabla 4.3: Regla PageReqRecByProxy en Java.

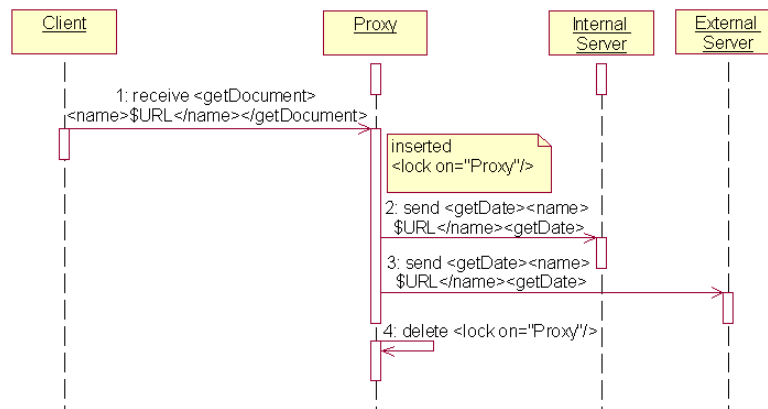


Figura 4.1: Diagrama de la regla PageReqRecByProxy

Regla 4: DateRespRecByProxy

La regla que se observa en la Tabla 4.4, sirve para modelar el comportamiento del proxy http cuando este recibe la respuesta de las peticiones hechas a los servidores interno y externo, en las que pide la fecha de la modificación más reciente del documento solicitado por el cliente.

Para que se active la regla, se deben recibir dos respuestas de las fechas, correspondientes al documento pedido por el cliente (tal documento se indica en la variable `$(Name)`), una fecha que provenga del servidor interno y otra que provenga del servidor externo.

La condición que se verifica es que la fecha del servidor externo esté vacía, es decir que se hayan enviado 8 ceros consecutivos, de ser así, se ejecuta el primer bloque contenido en los elementos `<do> </do>`. Dicho bloque se compone de una secuencia de tres pasos, el primer paso es enviar al servidor interno una orden en la que se indica que se borren todos los preparativos hechos para almacenar un documento `<dropDocument>` (esto por que cuando el servidor interno recibe la petición de una fecha y no la encuentra, inserta una etiqueta con los valores de los atributos vacíos suponiendo que como el documento no se encuentra almacenado se va a almacenar posteriormente). El segundo paso es depositar la llave en su lugar, para que otro usuario pueda ocupar el proxy (elemento `<insert><key></insert>`). El tercer paso es enviar al cliente un documento (etiqueta `<content>`) que contiene el mensaje “File Not Found *” que indica que el archivo que el cliente solicita no se localizó en el servidor Web al que se hizo la petición.

En esta regla hay una segunda condición, misma que se verifica sólo si la primera condición no se cumplió. En esta segunda condición se compara si las fechas del servidor externo e interno son iguales (lo que significa que el documento que se tiene en el servidor interno está actualizado), de ser así se ejecuta la acción que aparece en el segundo bloque `<do>`, la cual indica que se debe hacer una petición del documento solicitado por el cliente al servidor interno.

En caso de que ninguna de las dos condiciones anteriores se haya cumplido, se ejecutará la parte que se encuentra dentro de las etiquetas `<else> </else>`. Esta parte de la regla indica que se debe enviar al servidor externo una petición del documento que ha pedido el cliente. Esto se debe a que si no se cumplió la primera condición (que verifica que el servidor externo no tiene el documento) ni se cumplió la segunda condición (que verifica que las fechas del el servidor interno y externo son iguales) entonces sucede que las fechas son diferentes, y por ello el documento local no se encuentra actualizado. El diagrama de secuencia de esta regla se puede ver en la Figura 4.2

```

<rule name = "DateRespRecByProxy">
  <on>
    <received from = "InternalServer">
      <getDateResponse>
        <date>$DateInternalServer</date>
      </getDateResponse>
    </received>
    <received from = "ExternalServer">
      <getDateResponse>
        <date>$DateExternalServer</date>
      </getDateResponse>
    </received>
  </on>
  <if>
    <equal this = "$DateExternalServer" to = "00000000" />
  </if>
  <do><seq>
    <insert><lock/></insert>
    <send to = "InternalServer">
      <dropDocument><name>$Name</name></dropDocument>
    </send>
    <send to = "Client">
      <getDocumentResponse>
        <content>File Not Found *</content>
      </getDocumentResponse>
    </send>
  </seq></do>
  <if>
    <equal this = "$DateInternalServer" to = "$DateExternalServer" />
  </if>
  <do>
    <send to = "InternalServer">
      <getDocument><name>$Name</name></getDocument>
    </send>
  </do>
  <else>
    <send to = "ExternalServer">
      <getDocument><name>$Name</name></getDocument>
    </send>
  </else>
</rule>

```

Tabla 4.4: Regla DateRespRecByProxy.

```

public void run()
{
    for (;;)
    {
        getDateResponseInt = new getDateResponse();
        getDateResponseExt = new getDateResponse();
        if (uuid = (String) cInternalServer.received(getDateResponseInt) != null)
        {
            if (cExternalServer.received(uuid,getDateResponseExt) != null)
            {
                String name = getDateResponseInt.name;
                if (getDateResponseExt.date.equals("00000000"))
                {
                    cLock.insert(key);
                    DropDocument dropDocument = new DropDocument();
                    dropDocument.name = name;
                    cInternalServer.send(dropDocument);
                    GetDocumentResponse getDocumentResponse = new GetDocumentResponse();
                    getDocumentResponse.name = name;
                    getDocumentResponse.date = "00000000";
                    getDocumentResponse.content = "File not found";
                    cClient.send(getDocumentResponse);
                    cInternalServer.accept();
                    cExternalServer.accept();
                }
            }
            else
            if (getDateResponseInt.date.equals(getDateResponseExt.date))
            {
                GetDocument getDocument = new GetDocument();
                getDocument.name = name;
                cInternalServer.send(uuid,getDocument);
                cInternalServer.accept();
                cExternalServer.accept();
            }
            else
            {
                GetDocument getDocument = new GetDocument();
                getDocument.name = name;
                cExternalServer.send(uuid,getDocument);
                cInternalServer.accept();
                cExternalServer.accept();
            }
        }
        else
        {
            cInternalServer.reject();
        }
    }
}

```

Tabla 4.5: Regla DateRespRecByProxy en Java.

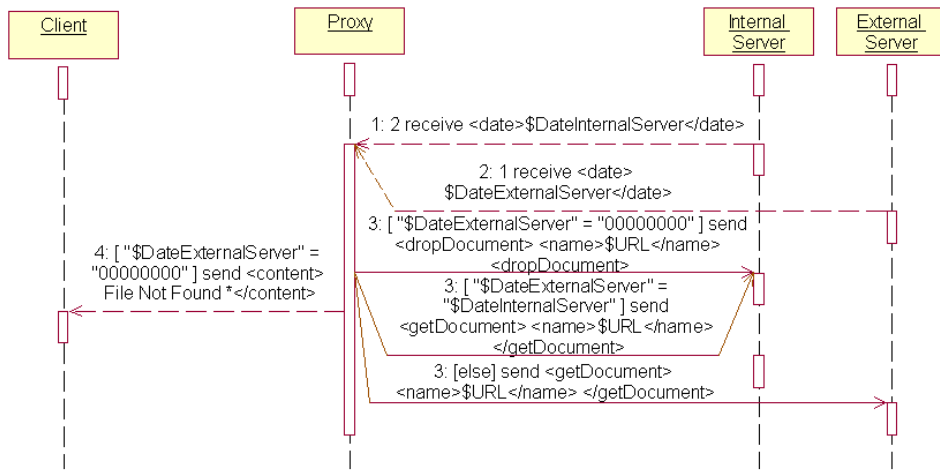


Figura 4.2: Diagrama de la regla DateRespRecByProxy

Regla 5: PageRespFromIntServRecByProxy

La regla mostrada en la Tabla 4.6 describe el comportamiento cuando el proxy http recibe una página Web desde el servidor Interno.

El evento que activa esta regla es recibir un documento (etiqueta <document>) desde el servidor interno (atributo **from** de la etiqueta <received>).

La condición en esta regla se omite puesto que se quiere ejecutar la acción directamente. La acción de la regla es devolver la llave del candado <lock> para que el proxy pueda atender otro cliente y enviar al cliente el documento que se ha recibido desde el servidor local <getDocumentResponse><content>\${Content}</content></getDocumentResponse>. El diagrama de secuencia de esta regla se muestra en la Figura 4.3.

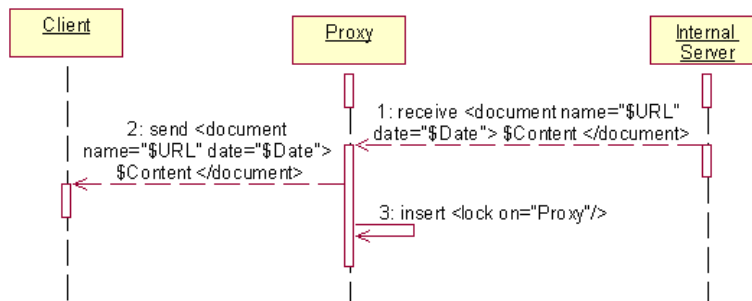


Figura 4.3: Diagrama de la regla PageRespFromIntServRecByProxy

Regla 6: PageRespFromExtServRecByProxy

La Tabla 4.8 muestra la regla que describe el comportamiento del proxy http cuando recibe una página Web desde el servidor externo.

El evento que activa esta regla es recibir un documento desde el servidor externo.

La condición en esta regla se omite. La acción de la regla es enviar al cliente el documento que

```

<rule name = "PageRespFromIntServRecByProxy">
  <on>
    <received from = "InternalServer">
      <getDocumentResponse>
        <name>$Name</name>
        <date>$Date</date>
        <content>$Content</content>
      </getDocumentResponse>
    </received>
  </on>
  <do><seq>
    <insert><lock></insert>
    <send to = "Client">
      <getDocumentResponse>
        <name>$Name</name>
        <date>$Date</date>
        <content>$Content</content>
      </getDocumentResponse>
    </send>
  </seq></do>
</rule>

```

Tabla 4.6: Regla PageRespFromIntServRecByProxy.

```

public void run()
{
  for (;;)
  {
    GetDocumentResponse getDocumentResponse = new GetDocumentResponse();
    if(uuid = (String) cInternalServer.received(getDocumentResponse) != null)
    {
      cLock.insert(key);
      cClient.send(getDocumentResponse);
      cInternalServer.accept();
    }
  }
}

```

Tabla 4.7: Regla PageRespFromIntServRecByProxy en Java.

```

<rule name = "PageRespFromExtServRecByProxy">
  <on>
    <received from = "ExternalServer">
      <getDocumentResponse>
        <name> $Name </name>
        <date> $Date </date>
        <content> $Content </content>
      </getDocumentResponse>
    </received>
  </on>
  <do> <seq>
    <send to = "Client">
      <getDocumentResponse>
        <name> $Name </name>
        <date> $Date </date>
        <content> $Content </content>
      </getDocumentResponse>
    </send>
    <send to = "InternalServer">
      <catchDocument>
        <name> $Name </name>
        <date> $Date </date>
        <content> $Content </content>
      </catchDocument>
    </send>
  </seq> </do>
</rule>

```

Tabla 4.8: Regla PageRespFromExtServerRecByProxy.

se ha recibido desde el servidor externo `<getDocumentResponse><content>$Content</content></getDocumentResponse>` y guardar en el servidor local el documento y un registro de dicho documento (`<catchDocument><name>$Name</name><date>$Date</date><content>$Content</content></catchDocument>`) para que almacene la información actualizada. Se puede ver el diagrama de secuencia de esta regla en la [Figura 4.4](#)

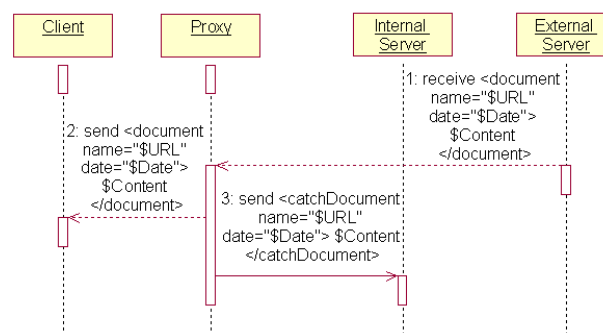


Figura 4.4: Diagrama de la regla PageRespFromExtServRecByProxy

Regla 7: ProxyAvailable

La regla mostrada en la [Tabla 4.10](#) sirve para comprobar que el proxy esté disponible para realizar la búsqueda de páginas. Funciona de la siguiente forma: cuando el proxy recibe una petición verifica si puede procesarla, esto lo hace consultando si está disponible la llave del


```

public void run()
{
    for (;;)
    {
        GetDocumentResponse getDocumentResponse = new GetDocumentResponse();
        if(uuid = (String) cExternalServer.received(getDocumentResponse) != null)
        {
            String name = getDocumentResponse.name;
            cClient.send(uuid,getDocumentResponse);
            CatchDocument catchDocument = new CatchDocument();
            catchDocument.name = name;
            catchDocument.date = getDocumentResponse.date;;
            catchDocument.content = getDocumentResponse.content;
            cInternalServer.send(uuid,catchDocument);
            cExternalServer.accept();
        }
    }
}

```

Tabla 4.9: Regla PageRespFromExtServRecByProxy en Java.

```

<rule name = "ProxyAvailable">
    <on>
        <received from = "InternalServer">
            <catchDocumentResponse>
                <request>Completed</request>
            </catchDocumentResponse>
        </received>
    </on>
    <do>
        <insert><lock></insert>
    </do>
</rule>

```

Tabla 4.10: Regla ProxyAvailable.

candado del proxy, de ser así hace las acciones necesarias para responder al cliente. De esta manera que si llega otra petición al proxy, y este se encuentra ocupado con una petición anterior (dado que la llave ya fué quitada), entonces el proxy ya no está disponible, por lo tanto hasta que se dé la respuesta al cliente anterior, el proxy coloca la llave para que otra petición pueda ser respondida.

La regla se activa cuando se recibe un mensaje cuya etiqueta `<catchDocumentResponse>` contiene la cadena "Completed" (dicho mensaje lo genera la regla 10) desde el servidor interno. Inmediatamente se ejecuta la acción, por lo que no se incluyó la parte de condición de la regla. La acción que se ejecuta es insertar (etiqueta `<insert>`) la llave que abre el candado del proxy (`<lock>`), con lo anterior se indica que el proxy está listo para reponder la siguiente petición. Se muestra el diagrama de secuencia de esta regla en la Figura 4.5

4.3.2. Reglas del Servidor Interno

El comportamiento del servidor interno se puede describir con cuatro reglas. La tarjeta CRC de esta clase se muestra en la Tabla 4.12.

```

public void run()
{
    for (;;)
    {
        CatchDocumentResponse catchDocumentResponse = new CatchDocumentResponse();
        if(uuid = (String) cInternalServer.received(catchDocumentResponse) != null)
        {
            if(catchDocumentResponse.request.equals("Completed"))
            {
                cLock.insert(key);
                cInternalServer.accept();
            }
            else
            {
                cInternalServer.reject();
            }
        }
    }
}

```

Tabla 4.11: Regla ProxyAvailable en Java.

Clase InternalServer	
Responsabilidades	Colaboraciones
<ul style="list-style-type: none"> ▪ Devolver la fecha de actualización mas reciente de un documento que está almacenado internamente ▪ Devolver el contenido de un documento que está almacenado internamente ▪ Almacenar el registro de un documento que incluya únicamente su nombre, dejando los otros campos vacíos. ▪ Complementar el registro cuyo nombre de documento ha sido previamente almacenado con su fecha y la ruta de localización en la base de documentos. Guardar el documento en la base de documentos. 	Proxy

Tabla 4.12: Tarjeta CRC de la clase InternalServer.

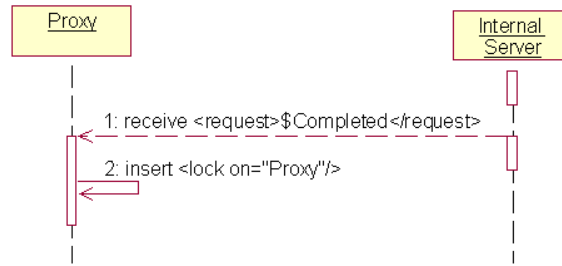


Figura 4.5: Diagrama de la regla ProxyAvailable

Regla 8: DateReqRecByIntServ

En la regla que se muestra en la Tabla 4.13 se puede ver el comportamiento del servidor interno cuando recibe la petición de una fecha.

La regla se activa cuando se recibe desde el Proxy la petición de la fecha de modificación más reciente (<getDate>) de un documento cuyo URL es el que corresponde con el almacenado en el elemento <name> en la variable \$Name.

Entonces se verifica si se tiene almacenado (<inserted into = "Cache">) el documento que corresponde al URL almacenado en la variable \$Name en base de documentos del servidor intern.

En caso de que la condición anterior se cumpla, se ejecuta la acción que se encuentra en el primer bloque <do> misma que especifica que se debe enviar al proxy la fecha que se encontró, y se enviará almacenada en la variable \$Date.

En caso de que no se cumpla la condición (lo cual implica que no se tiene almacenado el documento en el servidor interno) se ejecuta el bloque que se encuentra entre las etiquetas <else> y </else>. Este bloque consta de dos instrucciones, la primera envía al proxy una contestación indicando que la fecha del documento no se encuentra (esto lo hace enviando la etiqueta <date> con el contenido de 8 ceros), la segunda instrucción indica que se debe insertar en la *base de páginas* un registro de la página pedida, esto se hace para ahorrar tiempo suponiendo que se posteriormente se insertará el documento. El diagrama de secuencia de esta regla se muestra en la Figura 4.6

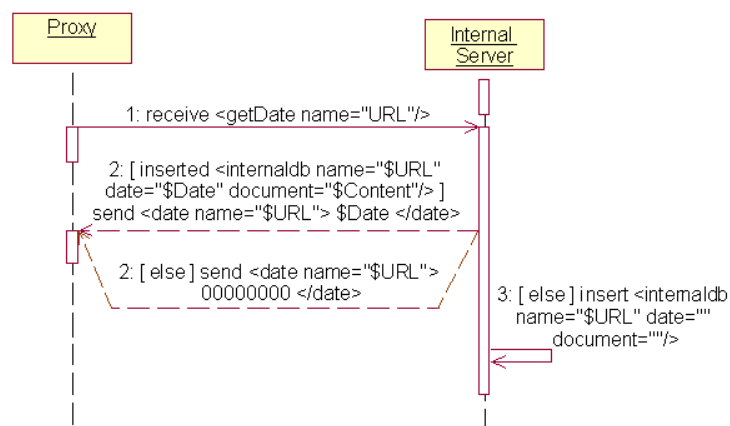


Figura 4.6: Diagrama de la regla DateReqRecByIntServ

```

<rule name = "DateReqRecByIntServ">
  <on><received from = "Proxy">
    <getDate><name>$Name</name></getDate>
  </received></on>
  <if><inserted into = "Regystry">
    <record>
      <name>$Name</name>
      <date>$._</date>
    </record>
  </inserted></if>
  <do><send to = "Proxy"><getDateResponse>
    <date>$Date</date>
  </getDateResponse></send></do>
  <else><seq>
    <send to = "Proxy"><getDateResponse>
      <date>00000000</date>
    </getDateResponse></send>
    <insert into = "Regystry"><record>
      <name>$Name</name>
      <date>$._</date>
    </record></insert>
  </seq></else>
</rule>

```

Tabla 4.13: Regla DateReqRecByIntServ.

```

public void run()
{
  for (;;) {
    GetDate getDate = new GetDate();
    if(uuid = (String) cProxy.received(getDate) != null){
      String name = getDate.name;
      DBRecord record = new DBRecord();
      record.name = name;
      GetDateResponse getDateResponse = new GetDateResponse();
      if(cRegistry.inserted(name, record)){
        getDateResponse.date = record.date;
        cProxy.send(uuid, getDateResponse);
        cProxy.accept();
      }
      else{
        getDateResponse.date = "00000000";
        cProxy.send(uuid, getDateResponse);
        record.name = name;
        record.date = "00000000";
        cRegistry.insert(name, record);
        cProxy.accept();
      }
    }
  }
}

```

Tabla 4.14: Regla DateReqRecByIntServ en Java.

```

<rule name = "PageReqRecByIntServ">
  <on>
    <received from = "Proxy">
      <getDocument><name>$Name</name></getDocument>
    </received>
  </on>
  <if>
    <inserted into = "Regystry"><record>
      <name>$Name</name>
      <date>$._</date>
    </record></inserted>
  </if>
  <do>
    <send to = "Proxy"><getDocumentResponse>
      <content>$Content</content>
    </getDocumentResponse></send>
  </do>
  <else><seq>
    <send to = "Proxy"><getDocumentResponse>
      <content>Proxy : File not found</content>
    </getDocumentResponse></send>
    <delete from = "Regystry"><record>
      <name>$Name</name>
      <date>$._</date>
    </record></delete>
  </seq></else>
</rule>

```

Tabla 4.15: Regla PageReqRecByIntServ.

Regla 9: PageReqRecByIntServ

En la Tabla 4.15 se muestra la regla que modela el comportamiento del servidor interno cuando recibe una petición de un documento.

El evento que activa la regla es que se reciba desde el Proxy la petición de un documento (etiqueta `<getDocument>`) que tiene el URL correspondiente al almacenado en el elemento `<name>`.

La condición que se verifica es que el documento esté almacenado (`<inserted into = "Regystry"><record>`) en la base de documentos del servidor local.

Si se cumple la condición anterior, se ejecuta la acción que se encuentra entre los bloques `<do>` y `</do>`. La acción específica que se debe enviar al proxy el contenido del documento en la variable `$Content`.

En el caso de que la condición no se cumpla, se ejecuta el bloque `<else>`, en ese bloque se realizan dos acciones, en la primera se envía la respuesta al proxy en una etiqueta `<content>` cuyo contenido es la cadena "Proxy: File not found". La segunda acción es borrar el documento que se insertó suponiendo que se iba a almacenar el documento posteriormente. Se puede ver el diagrama de secuencia de esta regla en la Figura 4.7

Regla 10: PageCachedByIntServ

La regla que se muestra en la Tabla 4.17, define el comportamiento del servidor interno cuando debe almacenar una página (su contenido propiamente dicho) y el registro de dicha página (los datos como su URL y la fecha de modificación) que no se tiene almacenada o bien que no está actualizada.

```
public void run()
{
  for (;;)
  {
    GetDocument getDocument = new GetDocument();
    if(uuid = (String) cProxy.received(getDocument) != null)
    {
      String name = getDocument.name;
      DBRecord record = new DBRecord();
      record.name = name;
      GetDocumentResponse getDocumentResponse = new GetDocumentResponse();
      if(cRegistry.inserted(name, record))
      {
        getDocumentResponse.date = record.date;
        getDocumentResponse.name = name;
        DBEntry entry = new DBEntry();
        entry.name = record.name;
        cDirectory.getEntry(name, entry);
        getDocumentResponse.content = entry.content;
        cProxy.send(uuid, getDocumentResponse);
        cProxy.accept();
      }
      else
      {
        getDocumentResponse.date = "00000000";
        getDocumentResponse.name = name;
        getDocumentResponse.content = "File not found";
        cProxy.send(uuid, getDocumentResponse);
        record.name = name;
        record.date = "00000000";
        cRegistry.delete(name, record);
        cProxy.accept();
      }
    }
  }
}
```

Tabla 4.16: Regla PageReqRecByIntServ en Java.

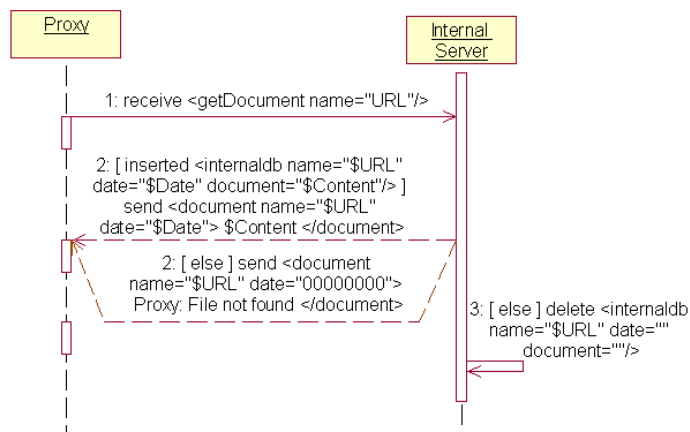


Figura 4.7: Diagrama de la regla PageReqRecByIntServ

La regla se activa cuando se recibe una etiqueta del tipo `<catchDocument>`, esta etiqueta indica que se debe almacenar una página nueva en la base de documentos, cuyo contenido está almacenado en la variable `$Content`, su fecha en la variable `$Date` y su URL en la variable `$Name`.

La codición de la regla indica que se debe verificar si ya se tiene almacenado un documento con el mismo URL pero con la fecha y contenido diferentes (variables anónimas `$_`).

Si la condición resulta verdadera, entonces se debe borrar el registro anterior (que no está actualizado), esto se hace mediante la etiqueta `<delete from = "Registry">`. Luego se debe insertar el registro que contiene la información actual (para ello se usa la etiqueta `<insert into = "Registry">`) y finalmente se envía al proxy la etiqueta `<request>Completed</request>`, misma que indica que la operación se ha efectuado satisfactoriamente. Esta última acción tiene como efecto activar la regla 7 antes descrita. Se puede ver el diagrama de secuencia de esta regla en la Figura 4.8.

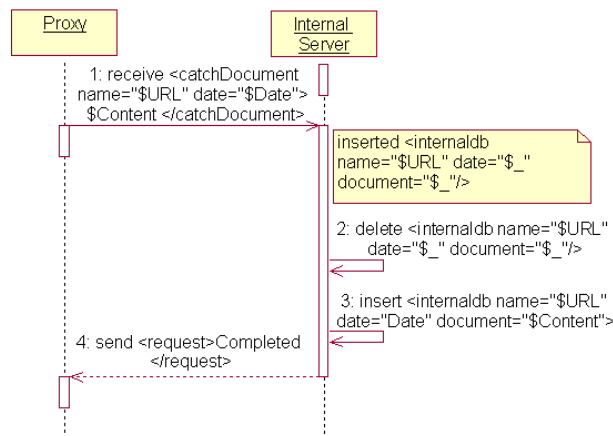


Figura 4.8: Diagrama de la regla PageCachedByIntServ

```
<rule name = "PageCachedByInternalServer">
  <on>
    <received from = "Proxy">
      <catchDocument>
        <name>$Name</name>
        <date>$Date</date>
        <content>$Content</content>
      </catchDocument>
    </received>
  </on>
  <if>
    <inserted into = "Registry"><record>
      <name>$Name</name>
      <date>$_</date>
    </record></inserted>
  </if>
  <do><seq>
    <delete from = "Registry"><record>
      <name>$Name</name>
      <date>$_</date>
    </record></delete>
    <insert into = "Registry"><record>
      <name>$Name</name>
      <date>$Date</date>
    </record></insert>
    <insert into = "Directory"><entry>
      <content>$Content</content>
    </entry></insert>
    <send to = "Proxy"><catchDocumentResponse>
      <request>Completed</request>
    </catchDocumentResponse></send>
  </seq></do>
</rule>
```

Tabla 4.17: Regla PageCachedByInternalServer.


```
public void run()
{
  for (;;)
  {
    CatchDocument catchDocument = new CatchDocument();
    if(uuid = (String) cProxy.received(catchDocument) != null)
    {
      String name = catchDocument.name;
      DBRecord record = new DBRecord();
      record.name = name;
      record.date = catchDocument.date;
      if(cRegistry.inserted(name, record))
      {
        DBRecord record2 = new DBRecord();
        record2.name = name;
        record2.date = "00000000";
        cRegistry.delete(record2.name, record2);
        cRegistry.insert(record.name, record);
        DBEntry entry = new DBEntry();
        entry.name = record.name;
        cDirectory.insert(entry.name, entry);
        CatchDocumentResponse catchDocumentResponse = new CatchDocumentResponse();
        catchDocumentResponse.request = "Completed";
        cProxy.send(uuid, catchDocumentResponse);
        cProxy.accept();
      }
      else
      {
        cProxy.reject();
      }
    }
  }
}
```

Tabla 4.18: Regla PageCachedByIntServ en Java.

```

<rule name = "PageDroppedByInternalServer">
  <on>
    <received from = "Proxy">
      <dropDocument><name>$Name</name></dropDocument>
    </received>
  </on>
  <if>
    <inserted into = "Regystry"><record>
      <name>$Name</name>
      <date>$.</date>
    </record></inserted>
  </if>
  <do><seq>
    <delete from = "Regystry"><record>
      <name>$Name</name>
      <date>$.</date>
    </record></delete>
    <send to = "Proxy"><dropDcoumentResponse>
      <request>Completed</request>
    </dropDcoumentResponse></send>
  </seq></do>
</rule>

```

Tabla 4.19: Regla PageDroppedByInternalServer.

Regla 11: PageDroppedByInternalServer

La Tabla 4.19 muestra la regla en que se describe el comportamiento del servidor interno cuando recibe la orden de borrar el registro de un documento. Esto se debe a se puede dar el caso en que el servidor interno reciba una petición para hacer las preparaciones para almacenar un documento y su registro, pero si el documento no se almacena por alguna razón, entonces se debe borrar este registro que se creó previamente. Por ejemplo en la Regla 4 se da este caso.

La regla se activa si el servidor interno recibe una etiqueta en la que se indica que debe borrar el registro (<dropDocument>).

La condición que se verifica es que se tenga almacenado el registro con el URL correspondiente a la variable \$Name.

Si la condición se cumple, entonces se borra el registro(etiqueta <delete from = "Regystry">) de la base de datos interna, y se envía al proxy la etiqueta <request>Completed</request> para indicar que la operación se realizó sin problemas, esta acción activa la regla 7. El diagrama de secuencia de esta regla se muestra en la Figura 4.9.

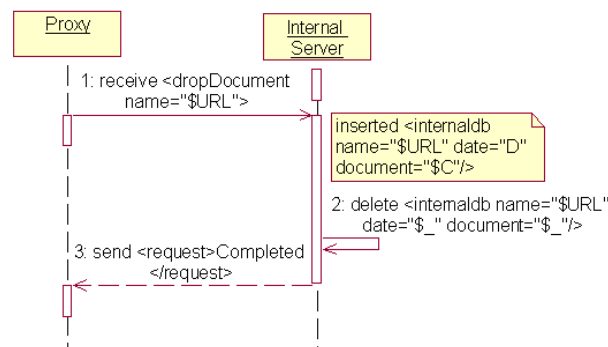


Figura 4.9: Diagrama de la regla PageDroppedByIntServ

```
public void run()
{
    for (;;)
    {
        DropDocument dropDocument = new DropDocument();
        if(uuid = (String) cProxy.received(dropDocument) != null)
        {
            String name = dropDocument.name;
            DBRecord record = new DBRecord();
            record.name = name;
            record.date = "00000000";
            if(cRegistry.inserted(name, record))
            {
                cRegistry.delete(name, record);
                DropDocumentResponse dropDocumentResponse = new DropDocumentResponse();
                dropDocumentResponse.request = "Completed";
                cProxy.send(uuid, dropDocumentResponse);
                cProxy.accept();
            }
            else
            {
                cProxy.reject();
            }
        }
    }
}
```

Tabla 4.20: Regla PageDroppedByIntServ en Java.

4.4. Análisis del sistema

Con el fin de validar el correcto funcionamiento del sistema, en esta sección se presentan cuatro casos, los cuales consideran todos los flujos que se pueden presentar en la ejecución del sistema.

1. Página no disponible en el proxy
2. Solicitud de página actualizada disponible en el proxy
3. Solicitud de página no actualizada disponible en el proxy
4. Página no existente

A continuación se describe detalladamente cada uno de estos casos.

4.4.1. CASO 1: Página no disponible en el proxy

En la Figura 4.10 se muestra el diagrama de secuencia del caso 1. El cliente solicita una página Web esta solicitud se envía al proxy cuando se ejecuta la acción de la **regla 1**.

El servidor proxy http esta escuchando todas la peticiones de los clientes, cuando recibe una petición se activa la **regla 3**. Se verifica la condición de que el proxy esté disponible, por lo que se debe encontrar la etiqueta <lock> que indica que la llave que abre el candado del proxy se encuentra disponible. Al ejecutarse la acción de la regla 3 se activan dos reglas, la regla 8 y la regla 12.

La **regla 8** se activa cuando se pide al servidor interno la fecha de una página Web para verificar si esta se encuentra almacenada en el servidor interno (que es propiamente dicho la caché del proxy http). No se cumple la condición de que el documento esté almacenado en la base de datos del servidor interno por lo que se ejecuta el bloque `<else>` de esta regla, la primera acción de dicho bloque es enviar al proxy una etiqueta `<date>` que tiene como contenido la siguiente cadena “00000000”. Con lo anterior se sabe que la página no se encuentra almacenada localmente. Además se inserta un registro con valores de atributos vacíos representando la no disponibilidad de información para que posteriormente cuando cuando se reciba una copia del servidor externo el documento se almacene.

La **regla 12** se activa cuando el servidor externo recibe una petición del proxy pidiendo la fecha de modificación de una página. Se cumple la condición de que el documento se encuentra almacenado en el servidor externo y por consecuencia se ejecuta la acción de la regla que se encuentra en el bloque `<do>` en donde se indica que se debe enviar la fecha de dicho documento al proxy usando la etiqueta `<date>`.

Las respuestas de las reglas 8 y 12 activan la **regla 4**, entonces se verifica la condición que se encuentra en el primer elemento `<if>` (en la que se verifica que el documento no se encuentra almacenado en el servidor externo), dado que no se cumple, se verifica la condición que está en el segundo elemento `<if>` (para cotejar si ambas fechas son iguales), y como tampoco se cumple. Entonces se deduce la página no se encuentra en el servidor externo o no está actualizada, por tanto se ejecuta la acción del elemento `<else>`. En esta parte se hace la petición del documento al servidor externo, activando así la regla 13.

El activarse en el servidor externo la **regla 13**, se verifica la condición de dicha regla para saber si el documento se encuentra almacenado en el servidor, la condición se cumple y se ejecuta la acción que es enviar al proxy el documento solicitado dentro de una etiqueta `<document>`. La recepción del documento desde el servidor externo activa la regla 6.

La **regla 6** carece de condición, por lo que ejecuta la acción que se conforma de dos ordenes, la primera es enviar al cliente el documento pedido (por lo que se activa la regla 2) y enviar al servidor interno la orden de que guarde el documento y un registro de dicho documento (se activa la regla 10).

Cuando se activa la **regla 2** se ejecuta la acción en la que el cliente escribe el contenido del documento solicitado.

Posteriormente la **regla 10** verifica que se tenga insertado un registro del documento que se pretende actualizar. Aunque este documento no se ha almacenado antes, se tiene un registro con atributos vacíos que se inserto como consecuencia de la regla 8. Dado que encuentra el registro, ejecuta la acción que se compone de tres pasos, primero borra el registro anterior (en este caso el de los atributos vacíos), luego inserta el documento y su registro nuevo, y por último envía al proxy la etiqueta `<request>` con la cadena “Completed”, para indicarle que se ha concluido la inserción del documento satisfactoriamente, lo que activa la regla 7.

La acción de la **regla 7** se ejecuta inmediatamente e inserta un documento con elemento raiz `<lock>` para indicar que la llave del candado está lista y por lo tanto el proxy está disponible.

4.4.2. CASO 2: Solicitud de página actualizada disponible en el proxy

En el diagrama de secuencia de la Figura 4.11, el cliente solicita una página (**regla 1**). La regla 1 activa la **regla 3**. El servidor proxy http recibe la petición, se verifica la condición a verdadera (que la llave del candado esté disponible `<lock>`) por lo que se sabe que el proxy está disponible. Se ejecuta la acción para hacer la consulta a la BD del servidor interno y verificar si la página se encuentra almacenada localmente, para ello hace una petición que activa la regla

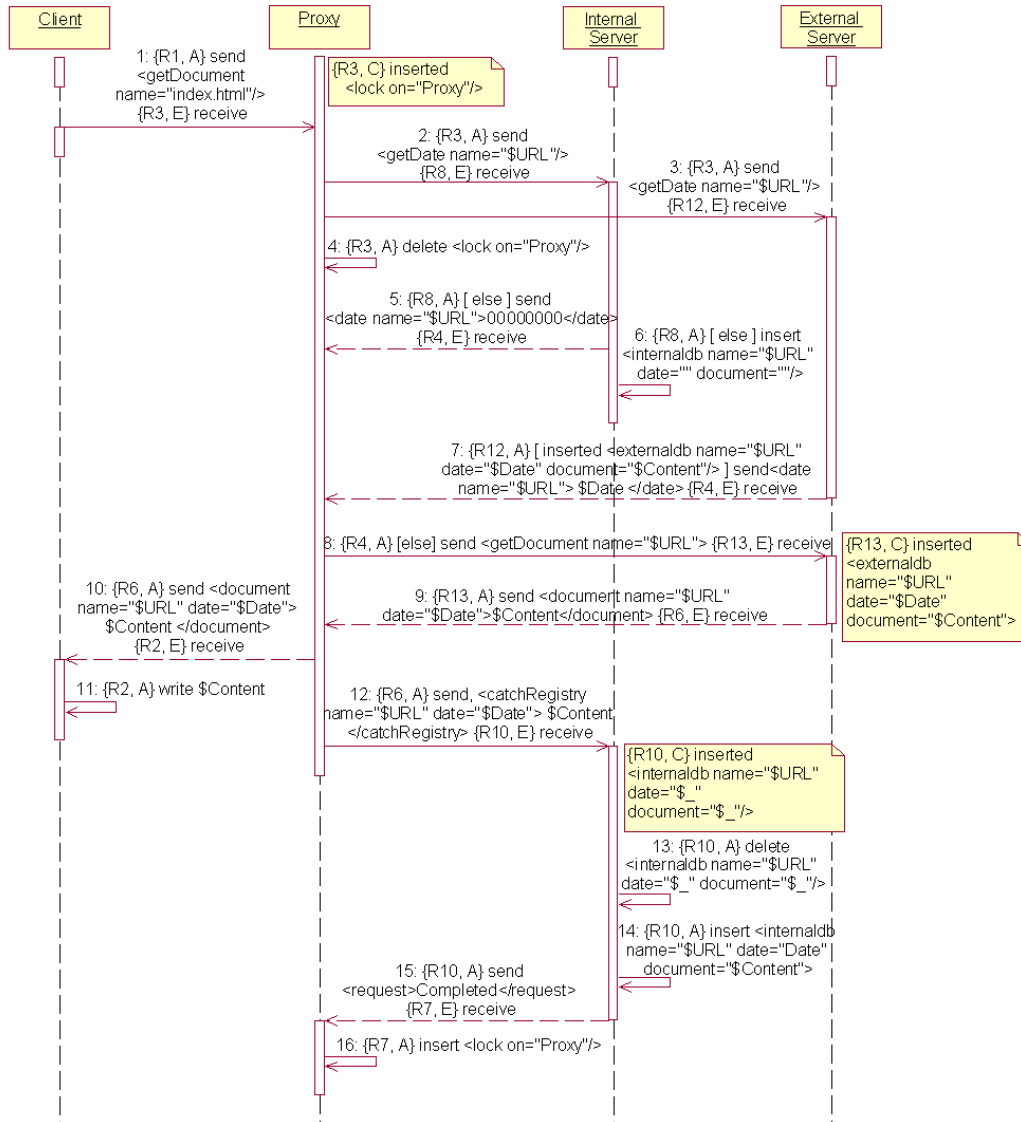


Figura 4.10: Caso 1: Página no disponible en el proxy

8. Además se lanza una petición al servidor externo para saber si tiene almacenado el documento lo que activa la regla 12.

Como resultado, al activarse la **regla 8** se verifica que efectivamente el documento esta almacenado en el caché del proxy entonces se ejecuta la acción que está en el primer elemento `<do>` enviando al proxy la fecha de actualización del documento dentro un elemento `<date>`.

Por otro lado la **regla 12**, verifica la condición y también se cumple (lo que implica que el documento está almacenado en el servidor externo), entonces se ejecuta la acción que se encuentra en el elemento `<do>` y mediante la cual se indica que se debe enviar la fecha de modificación más reciente del documento pedido.

Las respuestas de las reglas 8 y 12 activan la **regla 4**, se verifica la primera condición y no

se cumple. Se verifica la segunda condición (que la fecha del servidor interno y la del servidor externo sean iguales) y si se cumple, por lo que se ejecuta la acción del segundo elemento <do>. La acción indica que se debe pedir el documento al servidor interno debido a que se tiene el documento actualizado, tal petición activa la regla 9.

Al activarse la **regla 9** se verifica si el documento que se está pidiendo se encuentra almacenado, la condición se cumple y se ejecuta la acción que se encuentra en el elemento <do>, y es enviar el documento al proxy. Esta acción activa la regla 5.

La **regla 5** ejecuta la acción inmediatamente debido a que no tiene una condición que verificar. La acción indica que el proxy debe enviar al cliente el documento obtenido (lo que activa a regla 2) y además debe poner (elemento <insert>) la llave del candado para que el proxy quede disponible (<lock>).

La **regla 2** ejecuta su acción que es escribir el contenido del documento que se pidió.

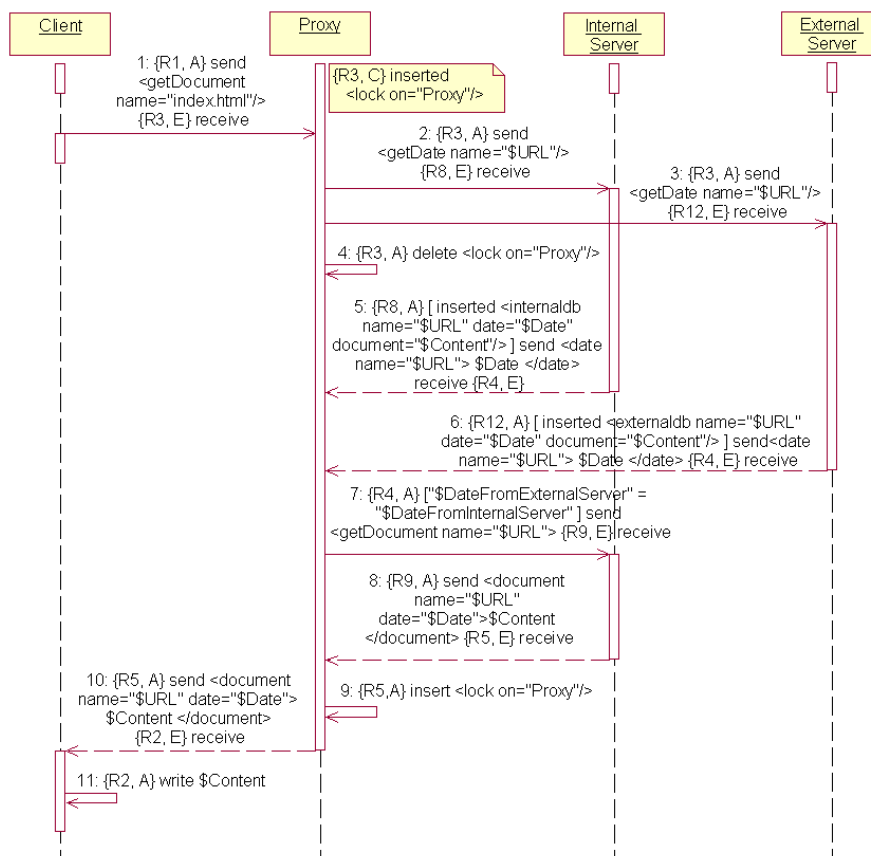


Figura 4.11: Caso 2: Solicitud de página actualizada disponible en el proxy

4.4.3. CASO 3: Solicitud de página no actualizada disponible en el proxy

Se puede ver el diagrama de secuencia de este caso en la Figura 4.12. La interacción empieza cuando el cliente solicita una página Web (**regla 1**). El servidor proxy http recibe la petición y se activa la **regla 3**. La condición de la regla 3 resulta verdadera (es decir el proxy está disponible

puesto que la llave del candado está en su sitio). Se ejecuta la acción de esta regla activando las reglas 8 y 12.

En la **regla 8** la condición se cumple por lo que el servidor interno envía al proxy la fecha del documento solicitado.

En la **regla 12** también se cumple la condición y se envía desde el servidor externo al proxy la fecha del documento.

La **regla 4** se activa con los resultados de las reglas 8 y 12. Se verifica la primera condición de la regla y no se cumple, se verifica la segunda condición y tampoco se cumple, por lo que se ejecuta la acción que se encuentra en el elemento `<else>`. La acción indica que se debe pedir el documento al servidor externo ya que el documento que se encuentra almacenado localmente no está actualizado. Como respuesta de esta acción se activa la regla 13.

La **regla 13**, verifica la condición para saber si el documento se encuentra almacenado en el servidor externo, la condición se cumple y se ejecuta la acción que es enviar al proxy el documento solicitado. La recepción del documento activa la regla 6.

La **regla 6** ejecuta la acción enviando al cliente el documento pedido (por lo que se activa la regla 2) y enviando al servidor interno la orden de que guarde el documento y un registro de dicho documento (se activa la regla 10).

Cuando se activa la **regla 2** se ejecuta la acción en la que el cliente escribe el contenido del documento solicitado.

La **regla 10** verifica que se tenga insertado un registro del documento que se pretende actualizar. Dado que encuentra el registro, ejecuta la acción que se compone de tres pasos, primero borra el registro anterior, luego inserta el documento y su registro nuevos, y por último envía al proxy un documento cuya raíz es el elemento `<request>` con la cadena “Completed”, para indicarle que se ha concluido la inserción del documento en el servidor interno satisfactoriamente, lo que activa la regla 7.

La **regla 7** ejecuta inmediatamente su acción debido a que no existe una condición que verificar. La acción de esta pone la llave del candado en su lugar para que el proxy quede disponible.

4.4.4. CASO 4: Página no existente

En el diagrama de secuencia de la Figura 4.13, el cliente solicita una página (**regla 1**). La regla 1 activa la **regla 3**. El servidor proxy http recibe la petición, se verifica la condición a verdadera (el proxy está disponible para responder esta petición) y ejecuta la acción pidiendo tanto al servidor interno como al externo la fecha de modificación del documento, lo que activa las reglas 8 y 12 respectivamente.

Como resultado, al activarse la **regla 8** se verifica la condición (que el documento esta almacenado en la caché del proxy), pero no se cumple y por ello se ejecuta el elemento `<else>` donde se envía al proxy un elemento `<date>` que tiene como contenido la siguiente cadena “00000000”. Con lo anterior se sabe que la página no se encuentra almacenada localmente. Además se inserta un registro con valores de atributos vacíos para que el documento se almacene cuando se tenga una copia del servidor externo.

Por otro lado la **regla 12**, verifica la condición (que el documento es almacenado por el servidor externo), sin embargo, puesto que no se cumple se ejecuta la acción que se encuentra en el elemento `<else>` la cual indica que se debe enviar el contenido de la fecha como una cadena de ocho ceros consecutivos.

Las respuestas de las reglas 8 y 12 activan la **regla 4**, se verifica la primera condición y se cumple, por lo que se ejecuta el primer elemento `<do>`. La acción indica que se debe enviar al servidor interno la orden de que borre el registro que preparó para almacenar el documento (se activa la regla 11) y debe enviar al cliente un documento cuyo elemento raíz es `<document>`, con

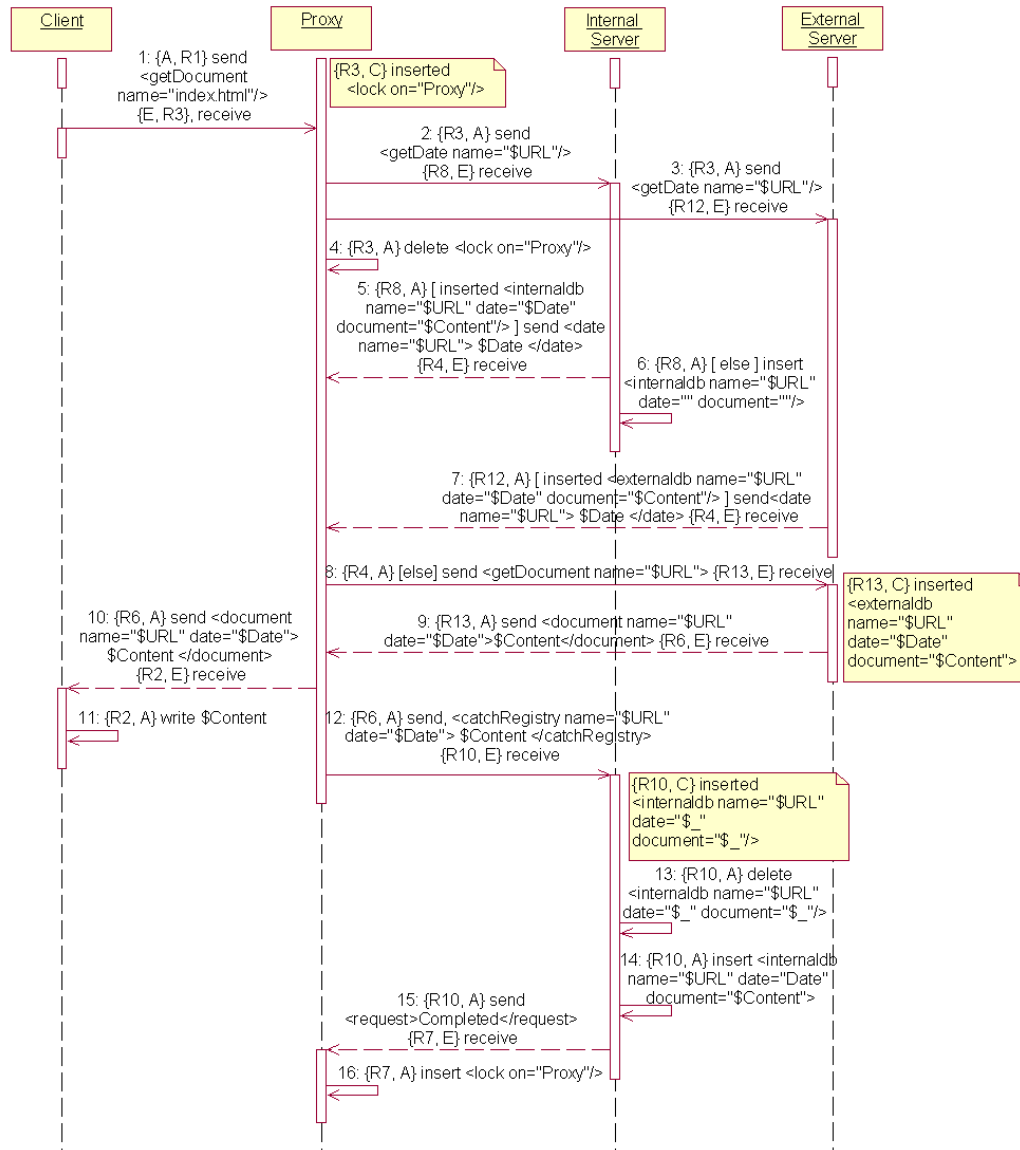


Figura 4.12: Caso 3: Solicitud de página no actualizada disponible en el proxy

el contenido “File not found” al cliente, para indicarle que la página que pidió no se localiza en el servidor externo (esto activa la regla 2).

Cuando se activa la **regla 11**, esta verifica que esté disponible un registro de la página en la base de datos, al encontrarlo se cumple la condición por lo que ejecuta la acción que es borrar el documento para despues enviar al proxy una etiqueta indicando que la acción se realizó satisfactoriamente, esta ultima acción activa la regla 7.

La **regla 2** ejecuta su acción que es escribir el contenido del documento que se pidió.

Cuando se ejecuta la acción de la **regla 7**, lo que se hace es poner la llave del candado en su lugar para que el proxy pueda responder a otras peticiones.

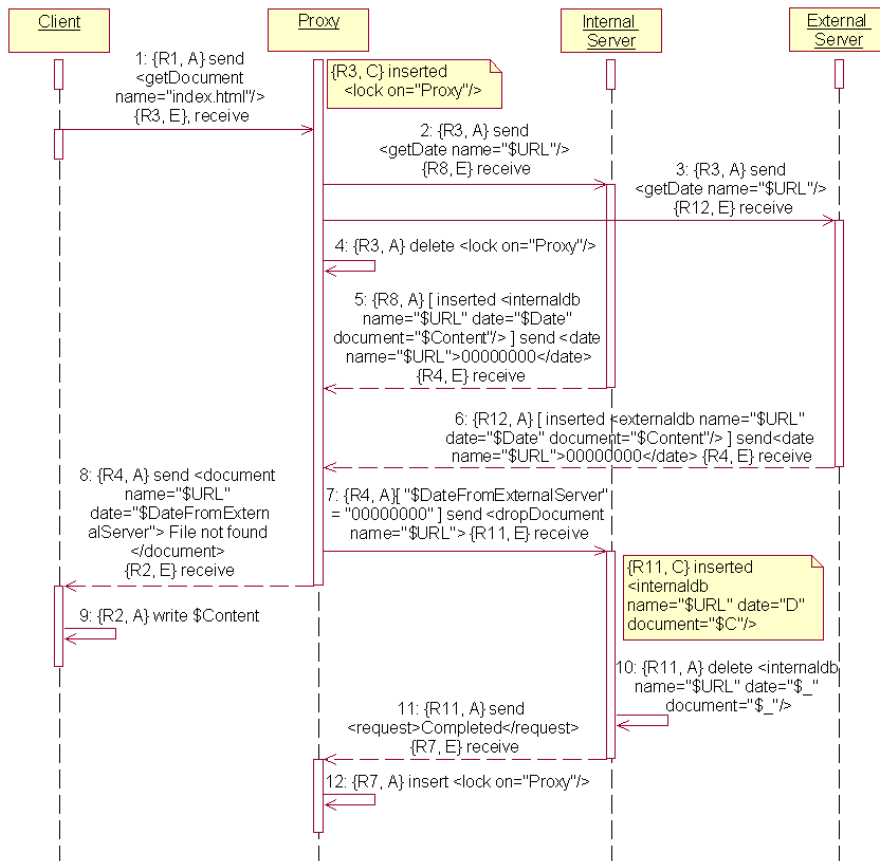


Figura 4.13: Caso 4: Página no existente

Capítulo 5

Implementación del lenguaje

5.1. Descripción del Administrador de Eventos

En este capítulo se describe la forma en que se implementó el lenguaje ADM. La programación se hizo en el lenguaje Java por que ofrece un conjunto de APIs para XML. Dichas APIs dan soporte a estándares de la industria (como DOM, WSDL y SOAP).

La interacción entre componentes distribuidos en ADM se basa en la abstracción conocida como *conector*. Un conector consiste de un espacio de objetos a los cuales se accede mediante un conjunto de operaciones sincronizadas (mutuamente exclusivas) que permiten intercambiarlos con otras aplicaciones u otros conectores. La interacción basada en conectores introduce la noción de *aceptación de mensajes*. Aunque relacionada con la recepción, la aceptación impone condiciones adicionales sobre el contenido del mensaje recibido para que sea consumido por alguna regla. La aceptación introduce un nivel de abstracción adicional a la recepción en el sentido en que los mensajes se reciben en el servidor al nivel de la capa de comunicaciones en tanto que los mensajes son aceptados en el conector al nivel de la capa de aplicaciones. En cuanto a la administración de los mensajes, un mensaje aceptado se elimina del conector, en tanto que otro no aceptado se mantiene elegible hasta que alguna regla lo acepte o su plazo expire y sea eliminado de acuerdo a alguna política de colección de basura. En este trabajo, la política seguida es no eliminarlos. En este trabajo se diseñaron y construyeron dos tipos de conectores básicos:

1. Conector para colecciones (bases de datos)
2. Conector para comunicaciones (mensajería SOAP)

A continuación describiremos cada uno de ellos.

5.2. Conector para comunicaciones (mensajería SOAP)

Las características que debe cumplir este conector son:

1. Envío y recepción de objetos
2. Envío y recepción de objetos asociados a un identificador
3. Envío asíncrono
4. Suscripción para recibir objetos específicos

5. Aceptación síncrona
6. Aceptación selectiva
7. Aceptación transaccional
8. Aceptación con conectores múltiples
9. Interacción tipo cliente-servidor

A continuación describiremos cada uno de ellos:

5.2.1. Envío y recepción de objetos

Constituye la forma más simple de comunicación. Los objetos deben poseer una estructura simple en la cual las variables de instancia admitan una representación serial. Estas condiciones permiten que los objetos puedan representarse en un formato compatible con XML tal como SOAP. De esta forma los objetos pueden intercambiarse entre aplicaciones remotas o mantenerse en medios de almacenamiento masivo. En la recepción, se indica la clase del objeto a la que debe pertenecer el objeto recibido como condición para su aceptación. En caso de que el objeto sea de la clase indicada, el objeto se acepta y la operación de recepción termina devolviendo el objeto aceptado; en otro caso, el objeto se mantiene en una lista de mensajes recientemente recibidos pero aún no aceptados. La lista de mensajes recientes indica que se consideran como eventos y que, por lo tanto, podrían ser aceptados por otra operación de recepción. La invocación síncrona a un procedimiento remoto corresponde al patrón envío-recepción en donde se indican las clases de los objetos enviados y recibidos, las cuales no necesariamente coinciden.

5.2.2. Envío y recepción de objetos asociados a un identificador

Esta forma de comunicación corresponde a aquellas situaciones en las que se asocia un identificador a una solicitud de manera que la respuesta correspondiente tenga asociada el mismo identificador. De esta forma, un servidor puede manejar las peticiones provenientes de diferentes solicitantes asegurando que todos ellos recibirán la respuesta que esperan. En este caso, la aceptación de una operación de recepción sucede cuando el objeto recibido corresponde con el identificador indicado y con la clase del objeto esperado. La invocación síncrona a un procedimiento remoto corresponde al patrón envío-recepción en donde se indican las clases de los objetos enviados y recibidos, las cuales no necesariamente coinciden. Generalmente, previo al envío se produce un identificador universal (no reproducible) que se usa tanto en el envío de la petición como en la recepción de la respuesta para corroborar la concordancia del identificador esperado.

5.2.3. Envío asíncrono

En el lado del cliente (el solicitante del servicio), el envío de un mensaje no detiene el curso de su ejecución. Antes de enviar al objeto, la infraestructura de comunicación lo convierte a su representación XML correspondiente de modo que pueda incluirse como el cuerpo del mensaje SOAP, incluyendo además la información de control correspondiente.

5.2.4. Suscripción por objetos específicos

Las operaciones de recepción gestionan una Suscripción ante el servidor por aquellos mensajes que desea recibir. La Suscripción permite asociar un tipo determinado de objeto con aquellas

reglas que las desean recibir de modo que cuando se produzca el mensaje del tipo adecuado, este sea enviado a uno de los subscriptores. La selección del subscriptor se realiza sobre la base de atender primero a aquella regla de quien se recibió primero la Suscripción. Esta política asegura la selección relativamente imparcial de reglas, condición que es importante observar en el diseño de sistemas distribuidos.

5.2.5. Recepción síncrona

En el servidor (el proveedor del servicio), la aceptación de un mensaje requiere necesariamente de su previa recepción. Por tal razón, si el mensaje está disponible previamente a la recepción, entonces el curso de la ejecución no se detiene y el mensaje puede aceptarse. Por otra parte, si el mensaje no está disponible, entonces el curso de la ejecución se detiene hasta que el servidor reciba un mensaje aceptable.

5.2.6. Aceptación de eventos compuestos

En una regla se pueden aceptar objetos de diferentes tipos que provienen de diferentes conectores. Esto permite aceptar una secuencia de mensajes de acuerdo a un protocolo multipartita de comunicación. Por simplicidad, los eventos compuestos se reconocen sobre un horizonte infinito por lo que las aplicaciones deben definir sus propias heurísticas para eliminar mensajes que no son más de interés. La separación entre las operaciones de recepción y de aceptación simplifica la tarea de reconocimiento de eventos compuestos. Para reconocer, por ejemplo, un evento compuesto formado solamente por un evento elemental, a la operación de recepción le debe suceder de inmediato una operación de aceptación. Por otra parte, para reconocer un evento compuesto formado por una secuencia de eventos elementales, a la secuencia de operaciones de recepción le debe suceder la operación de aceptación. Sin embargo, el reconocimiento de eventos compuestos que involucra a múltiples conectores conlleva diversos problemas relacionados por la competencia no coordinada de las instancias de reglas durante la recepción de mensajes.

5.2.7. Aceptación selectiva

Las reglas pueden competir por la aceptación de mensajes cuando se combinan bajo un esquema de composición alternativa. La composición alternativa describe la selección de un comportamiento de entre varios posibles. ADM utiliza el modelo de *compromiso en la selección* (*committed choice*), en donde, la decisión sobre la alternativa a seguir se toma sobre la base de adoptar aquella que primero realice una aceptación. Una vez tomada la decisión, las restantes alternativas se descartan. La alternativa seleccionada puede confirmarse o rechazarse más adelante de acuerdo al modelo de aceptación transaccional que explica a continuación.

5.2.8. Aceptación transaccional

La aceptación competitiva en donde intervienen múltiples conectores puede dar lugar a condiciones de interferencia mutua entre reglas. Para evitar los problemas derivados de la interferencia, ADM proporciona *aceptación transaccional*. La aceptación transaccional evita que varias reglas interfieran entre sí cuando alguna regla reciba mensajes que otras requieran. Los conectores ofrecen las operaciones `accept()` y `reject()` para aceptar o rechazar todas las operaciones de recepción aplicadas recientemente. En ADM, la operación `accept()` se invoca cuando la condición de la regla ha sido verificada; en otro caso, se invoca la operación `reject()`. Con este esquema, se asegura que se cumplen las propiedades transaccionales de atomicidad (solamente una regla a la

vez), consistencia (solamente después de verificar la condición), integridad (solamente operaciones aceptadas) y durabilidad (una vez aceptadas se hacen permanentes los cambios).

5.2.9. Interacción cliente-servidor

Los conectores adoptan la arquitectura cliente-servidor de los *sockets*. En el servidor, la infraestructura de comunicación de ADM crea automáticamente un conector que recibe las peticiones SOAP dirigidas a él. En el cliente, las reglas de la aplicación deben crear un conector que indique la dirección del servidor. Generalmente, los conectores se identifican mediante un URL que indica su localización en Internet y que incluye un identificador local de conector en el servidor.

5.2.10. Estructuras de datos

Un conector consiste de varias estructuras de datos que se pueden clasificar en simples y compuestas.

Estructuras de datos básicos

Las estructuras de datos básicos constituyen tipos predefinidos por el entorno computacional donde se ubica ADM.

- **String:** Arreglo inmutable de caracteres consecutivos.
- **Type:** Designación de un conjunto de objetos que se caracterizan por las estructuras de datos y las operaciones que comparten. En general, el tipo corresponde con la clase de los objetos.
- **Rule:** Unidad de ejecución de un programa que representa a una regla como entidad activa (proceso o *thread*).
- **Synchronizer:** Monitor que asegura el acceso exclusivo a regiones críticas de código y datos.

Estructuras de datos compuestas

- **Encabezado del mensaje (Head).** Estructura de control de un mensaje. El encabezado del mensaje incluye un identificador global único (UUID) del mensaje así como información sobre su conector de origen. La información de control se utiliza durante el envío y la recepción del mensaje.
- **Cuerpo del mensaje (Body).** Estructura de datos de un mensaje. El cuerpo del mensaje es la clase base de la que se deriva cualquier tipo de objeto. Antes de ser enviado físicamente por un *socket* TCP/IP (o almacenarse como registro en una tabla de una base de datos), el cuerpo del mensaje debe convertirse a una forma plana (*serializable*) mediante un proceso conocido como *marshalling*, realizando el proceso inverso una vez que ha sido recibido (o recuperado de un registro de una tabla) mediante el proceso inverso conocido como *unmarshalling*.
- **Saliente (Outgoing).** Lista de mensajes por enviar. La lista está formada por pares de instancias de estructuras de datos **Head** y **Body**, las cuales representan respectivamente al componente con información de control de mensajería (encabezado del mensaje **Head**) y al componente de datos de la aplicación (cuerpo del mensaje **Body**). Formalmente:

$$Outgoing = [(Head, Body)]$$

Para cada elemento de la lista `Outgoing`, las instancias de `Head` y `Body` se serializan al formato SOAP antes de enviarse a la dirección asociada al conector. Los elementos de la lista se procesan de acuerdo a la política FIFO de extraer primero (`removeFirst()`) al que arriba primero (`addLast()`).

- **Entrante (Incoming).** Lista de mensajes recibidos. Al igual que `Outgoing`, esta lista está formada por pares de instancias de estructuras de datos `Head` y `Body`, que contienen información de control de mensajería y de datos de la aplicación.

$$Incoming = [(Head, Body)]$$

Para cada elemento de la lista `Incoming`, las instancias de `Head` y `Body` se deserializan del mensaje SOAP que se recibe de la dirección asociada al conector. Los elementos de la lista se procesan de acuerdo a la política FIFO usual de extraer primero (`removeFirst()`) al que se inserta primero (`addLast()`).

- **Recibido (Received).** Tabla de objetos recibidos (`Body`) clasificados por su tipo (`Type`) y por su identificador (`Head`). Esta organización permite seleccionar de entre los objetos recibidos aquellos que poseen el tipo indicado en la operación de recepción. Cuando además el objeto tiene un identificador asociado, entonces el identificador se usa para determinar si dicho objeto ya ha sido recibido.

$$Received = Type \rightarrow (Head \rightarrow Body)$$

En el caso en que la aplicación no le asigna al objeto un identificador, ADMRM le asigna un identificador generado de manera que se asegure su unicidad.

- **Respaldo (Backup).** Lista de respaldo de los mensajes que han sido aceptados pero no confirmados o rechazados. Cada vez que un objeto es aceptado, este es automáticamente puesto en respaldo en esta lista, en donde permanecerá hasta que sea confirmada su aceptación por una operación `accept()` o rechazado por una operación `reject()`.

$$Backup = [(Head, Body)]$$

La operación `accept()` simplemente descarta todos los elementos de esta lista, mientras que la operación `reject()` los hace de nuevo disponibles para que puedan ser considerados en otras operaciones de recepción.

- **Subscriber (Subscriber).** Subscriber que solicita y espera recibir un objeto de tipo determinado. ADMRM asocia a cada regla activa un *thread* en ejecución de manera que el subscriber es la entidad que espera recibir al objeto. Generalmente, el subscriber debe esperar hasta que el objeto esté disponible, suspendiendo con ello su actividad. Una vez disponible, el subscriber es notificado con lo cual se reanuda la ejecución de la operación de recepción. Un subscriber consiste de un par (`Head,Body`) que representa al objeto (`Body`) e identificador (`Head`) de interés para el subscriber, de un apuntador al grupo de competidores al cual pertenece el subscriber y, finalmente, de un monitor `Synchronizer` que coordina el curso de la ejecución de la regla.

$$\text{Subscriber} = (\text{pair} : \text{Pair}, \text{contention} : \text{Activity}, \text{sync} : \text{Synchronizer})$$

El sincronizador es un monitor que asegura el acceso restringido a un objeto mediante un candado. El invariante del sincronizador establece que solamente un *thread* a la vez puede permanecer activo ejecutando los métodos de la clase a la que pertenece el objeto, de manera que otros *threads* que desean ejecutar el código se mantienen en una lista de espera. El sincronizador ofrece operaciones que permiten coordinar las interacciones de los *threads* como `wait()`, `notify()` y `notifyAll()`.

- **Suscripción (Subscription).** Tabla que agrupa a los subscriptores de acuerdo al tipo de información que solicitan. Todos los subscriptores que demuestran interés por un objeto de un tipo determinado se agrupan en una lista conocida como **Suscripción**. Así, la estructura suscripción es una función parcial del tipo de objetos de interés con la lista de subscriptores que los requieren.

$$\text{Subscription} = \text{Type} \rightarrow [\text{Subscriber}]$$

Los subscriptores en la lista están organizados de acuerdo al orden de la aparición de su solicitud de manera que se ofrezca imparcialidad en el otorgamiento del servicio.

- **Transacción (Transaction).** Tabla que asocia a cada *thread* en ejecución con un subscriptor y a este con la lista de respaldo que será usada en una operación de recepción transaccional. Cuando un solicitante se suscribe a un servicio, el *thread* del solicitante se usa para identificar a su subscriptor ($\text{Thread} \rightarrow \text{Subscriber}$). A su vez, a cada subscriptor se le asocia una lista de respaldo de da-tos utilizados por una regla ($\text{Subscriber} \rightarrow \text{Backup}$). En caso de que no se le haya asignado uno, entonces se le asigna uno nuevo y se registra en las listas de suscripciones. Una vez identificado el subscriptor, se recupera la lista de respaldo asociada.

$$\text{Transaction} = \text{Thread} \rightarrow \text{Subscriber} + \text{Subscriber} \rightarrow \text{Backup}$$

La tabla de transacción permite recuperar el subscriptor asociado con el *thread* en ejecución, de manera que se pueda reanudar el curso de la ejecución cuando este se encuentre suspendido, principalmente cuando faltan objetos aceptables.

Una vez descritas las principales estructuras de datos del conector de mensajería, a continuación describiremos las clases y los métodos que las utilizan.

Estructuras de control

Las estructuras de control consisten de un conjunto de procesos que se combinan para formar procesos compuestos mediante construcciones estructuradas de alto nivel. Las reglas activas se consideran como actividades elementales.

- **Actividad (Activity)**

Representa la clase de entidades con comportamiento relativamente autónomo generalmente implementado como procesos ligeros o *threads*. Las reglas activas y los conectores constituyen actividades elementales (i.e., subclases de la clase *Activity*), mientras que la composición de reglas forma grupos estructurados llamados actividades colectivas (*CollectiveActivity*).

- Constructores
 - `Activity(String name)`
 - `Activity(String name, CollectiveActivity group)`
- Métodos
 - `setGroup(CollectiveActivity group)`. Establece el grupo al que pertenece la actividad
 - `getGroup() : CollectiveActivity`. Indica el grupo al que pertenece la actividad
 - `setSelection(Object act)`. Establece la actividad seleccionada
 - `getSelection()`. Indica cuál es la actividad seleccionada
 - `undoSelection(Object act)`. Cancela la actividad seleccionada
 - `started()`. Inicia la actividad
 - `stopped()`. Termina la actividad
- **Actividad Colectiva (CollectiveActivity)**

Describe colecciones o grupos estructurados de actividades. Las colecciones forma estructuras de control clásicas en concurrencia como actividades que se realizan de manera secuencial, paralela y alternativa (choice). `CollectiveActivity` es una clase derivada de `Activity` y, a su vez, es la clase base para cada una de las clases `Sequential`, `Parallel` y `Alternative`. El significado (semántica operacional) de cada una de ellas es el descrito en el modelo de programación de ADM.

 - Constructores:
 - `CollectiveActivity(String name, Activity[] act)`
 - `CollectiveActivity(String name, Activity[] act, CollectiveActivity group)`
- **Secuencial (Sequential)**

Ejecuta una colección de actividades en forma secuencial. En una actividad colectiva secuencial, la ejecución de una actividad no inicia hasta que no haya terminado la ejecución de la actividad anterior. Una actividad secuencial inicia su ejecución cuando inicia la ejecución de la primera actividad de la colección y termina su ejecución cuando haya terminado la ejecución de la última actividad de la colección. Esta clase se deriva de `CollectiveActivity`.

 - Constructores:
 - `Sequential(String name, Activity[] act)`
 - `Sequential(String name, Activity[] act, CollectiveActivity group)`
- **Paralela (Parallel)**

Ejecuta una colección de actividades en forma paralela. Una actividad colectiva paralela inicia cuando todas las actividades de la colección inician su ejecución simultáneamente y termina cuando la última actividad (la más lenta) termina su ejecución. Esta clase se deriva de `CollectiveActivity`.

 - Constructores:
 - `Parallel(String name, Activity[] act)`
 - `Parallel(String name, Activity[] act, CollectiveActivity group)`

■ Selectiva (Alternative)

Ejecuta solamente una actividad de una colección de actividades. Esta construcción ofrece mecanismos de exclusión mutua entre las actividades de la colección de manera que a lo más una actividad pueda ejecutarse. La regla de selección es indeterminista y generalmente se elige la primera de la colección que recibe un mensaje aceptable. Una actividad colectiva alternativa inicia y termina su actividad cuando así lo hace la actividad seleccionada. Esta clase se deriva de `CollectiveActivity`.

- Constructores:
 - `Alternative(String name, Activity[] act)`
 - `Alternative(String name, Activity[] act, CollectiveActivity group)`

■ SOAPConnector

Representa un espacio de mensajes que son introducidos cuando se reciben y eliminados cuando finalmente son aceptados. El envío de un mensaje es una operación fundamental observable, es decir, producen un evento que es notificado a todo aquel *thread* que se haya suscrito mediante una operación de recepción. El envío asíncrono de mensajes requiere que el conector sea una entidad activa encargada de enviar físicamente todos los mensajes recolectados por la operación `send()`. La recepción síncrona causa que todo proceso se suscriba para que sea notificado tan pronto como un mensaje aceptable se encuentre disponible. El conector ofrece un mecanismo flexible de aceptación o rechazo de mensajes al separar las operaciones `accept()` y `reject()` de la operación de recepción `receive()`. Esto permite que se puedan analizar secuencias completas de mensajes para examinar sus contenidos antes de que sean aceptados (y por consiguiente, completamente removidos) o bien rechazados (y por lo tanto, disponibles para otras reglas).

- Constructor. `SOAPConnector(String url)`. Crea un nuevo conector que guarda conexión con el conector remoto situado en la dirección dada por el url. Los mensajes enviados por el conector local serán recibidos en el conector remoto y, recíprocamente, todo mensaje de respuesta enviado por el conector remoto será recibido en el conector local correspondiente.
- Método `run()`. Define el comportamiento del conector. El comportamiento se divide en dos fases: (i) de envío de objetos al conector remoto instalado en la dirección dada por el url y (ii) de recepción de objetos provenientes de dicho conector remoto. En la primera fase, el conector extrae cada objeto a ser enviado junto con la información de control correspondiente de la cola `outgoing` siguiendo una política FIFO. Antes de ser enviado, el objeto es convertido a una forma plana (codificado en XML) mediante la operación de `marshall()`. Entonces, se ensambla un mensaje SOAP cuyo encabezado se forma con la información de control y cuyo cuerpo se forma con la representación plana del objeto. Una vez ensamblado el mensaje, el conector envía el mensaje al *url* indicado para quedarse a continuación en espera por la respuesta. Cuando la respuesta eventualmente se recibe, el mensaje es convertido a objeto mediante la operación `unmarshall()`. Después de ser reconstruido, el objeto se clasifica de acuerdo a su tipo en la tabla `received`. Si es el primer objeto recibido de este tipo, entonces se crea una subtabla para agrupar a todos los objetos del mismo tipo. Esta organización induce una partición en `received` de acuerdo al tipo de los objetos. Por otra parte, es posible que se reciba más de un objeto con el mismo `uuid`, un problema que puede originarse en la aplicación. En este caso, todos los objetos que comporten el mismo `uuid` se agrupan

en una lista. Una vez que el objeto ha sido recibido junto con la información de control correspondiente, el par es puesto en la lista de mensajes entrantes `incoming`, con lo cual inicia la segunda fase.

En la segunda fase, el conector analiza, para cada tipo de objeto de la lista `incoming`, si existe una suscripción para recibir un objeto de ese tipo. Si no existe una suscripción para ese tipo de objeto, entonces se considera al siguiente objeto de la lista `incoming`. En caso de que no existiera alguna suscripción para cualquiera de los objetos de la lista `incoming`, el conector queda en espera. En caso de que existiera alguna suscripción, entonces se determina la naturaleza del suscriptor:

- El suscriptor entra en una competencia (colección de actividades alternativas) en donde ningún suscriptor ha sido seleccionado aún. En este caso, el suscriptor es considerado antes que sus competidores y, por tal razón, es seleccionado. Esta medida garantiza la mutua exclusión con otros suscriptores.
- El suscriptor entra en una competencia en donde ya existe un seleccionado. En este caso, si el suscriptor no es el seleccionado, entonces abandona la competencia y se agrega de nuevo a la lista de suscripción para su futura reconsideración.
- En cualquier otro caso, el suscriptor es seleccionado.

Al suscriptor seleccionado se le asigna el objeto, reanudando a continuación el desarrollo de sus actividades las cuales quedaron suspendidas por su invocación a `receive()`. De esta forma, termina la segunda fase habiendo obtenido el suscriptor un objeto del tipo esperado.

- Método `send(Head, Body)`. Ensambla un par (head,body) y lo inserta al final de la cola de espera de objetos a ser enviados `outgoing`.
- Método `send(Body)`. Ensambla un par (head,body) y lo inserta al final de la cola de espera de objetos a ser enviados `outgoing`. En este caso, el encabezado no contiene un identificador universal para el objeto que se encuentra en el cuerpo.
- Método `receive(Head, Body)`. Recibe un objeto aceptable (`Body`) junto con la información de control relacionada con él (`Head`). Esta operación se realiza en tres fases: (i) preparación, (ii) espera y (iii) transferencia. En la primera fase (de preparación) se realizan las siguientes acciones:
 - Se registra el suscriptor (*thread* en ejecución) en la tabla de transacciones (*transaction*) usando el identificador del thread como clave de acceso. Si el suscriptor ya hubiera sido registrado en la tabla de transacciones, simplemente se recupera su valor.
 - Se determina si el suscriptor pertenece a un grupo en competencia (actividades alternativas). En tal caso, el suscriptor conserva una copia del grupo al que pertenece siempre que en el grupo aún no haya un suscriptor seleccionado.
 - Se asocia una lista de respaldo en caso del que el suscriptor no posea alguna. La lista de respaldo se usa para guardar los datos recibidos por las operaciones `receive()`.
 - Se registra el suscriptor en una lista de suscripción (*subscription*) en caso de que no haya sido registrado aún de acuerdo al tipo del objeto que busca obtener. Si ya se encontraba registrado en una lista de suscripción, entonces se remueve y se inserta al final.

Al final de la primera fase, el suscriptor ha quedado debidamente registrado en la lista de suscripción apropiada. En la segunda fase (de espera), el suscriptor espera a

que el conector le notifique que un objeto aceptable se encuentra disponible. Esta fase causa que el suscriptor suspenda sus actividades. En la tercera fase (de transferencia), el suscriptor reanuda su actividad al recibir la notificación de que un objeto aceptable le será asignado. El objeto asignado se remueve de la lista de objetos entrantes (*incoming*) a la vez que se agrega en la lista de respaldo (*backup*). Finalmente, el objeto (cuerpo del mensaje) recibido se copia en el objeto provisto como parámetro, mientras que la información de control (encabezado del mensaje) se regresa como resultado de la invocación a `receive()`.

- Método `receive(Body)`. Esta operación es como la anterior versión de `receive()` excepto que no se conoce la información de control. Este método se usa cuando se desea recuperar un objeto de un tipo dado sin imponer las condiciones adicionales dadas en el encabezado.
- Método `accept()`. Causa la aceptación de los mensajes recibidos de manera que ningún otro suscriptor los pueda recibir. Lo anterior se consigue eliminando el contenido de la lista de respaldo (*backup*). Además, este método cancela al suscriptor seleccionado en el grupo de modo que otros suscriptores puedan reanudar la competencia.
- Método `reject()`. Causa el rechazo de los mensajes recibidos de modo que otros suscriptores (incluido el suscriptor que emite `reject()`) tengan la posibilidad de recibirlos después. Los mensajes recibidos acumulados en la lista de respaldo (*backup*) son agregados a la tabla de recibidos y a la lista de objetos entrantes.

5.3. Conector para colecciones (bases de datos)

El conector para colecciones posee las siguientes características:

1. Inserción, eliminación y búsqueda de objetos
2. Inserción, eliminación y búsqueda de objetos con identificador
3. Inserción y eliminación (generadora de eventos)
4. Inserción y eliminación silenciosa
5. Suscripción para recibir objetos de un tipo dado
6. Aceptación síncrona
7. Aceptación selectiva
8. Aceptación transaccional
9. Aceptación de eventos compuestos
10. Interacción tipo cliente-servidor

A continuación describiremos estas características brevemente.

5.3.1. Inserción, eliminación y búsqueda de objetos

Estas son las operaciones fundamentales definidas sobre una colección de datos (base de datos). Antes de la inserción, los objetos se convierten a una representación plana antes de ser almacenados como registros en una tabla de una base de datos. Después de la búsqueda, los objetos se recuperan a partir de la información obtenida del resultado de la consulta. Cuando se declara el conector, se asocia a una tabla de una base de datos mediante un nombre que es reconocido por un puente para la base de datos como la ODBC-JDBC. De esta forma, los objetos encuentran en la base de datos física un medio de almacenamiento persistente.

5.3.2. Inserción, eliminación y búsqueda de objetos con identificador

Las operaciones fundamentales se pueden realizar usando un identificador de objeto que sirve como llave primaria. El identificador permite almacenar y recuperar un objetos en tablas indexadas por su identificador (llave primaria) para su rápida localización.

5.3.3. Inserción y eliminación de objetos (generadora de eventos)

Las operaciones de inserción y eliminación producen eventos observables. Las operaciones se realizan sobre estructuras de datos internas antes de que tengan efecto sobre las bases de datos que usan como almacenamiento estable. Este diseño facilita la administración de las operaciones ya que estas se pueden procesar antes de que modifiquen el contenido de la base de datos. Las operaciones finalmente tienen efecto durante su aceptación o rechazo. Una vez almacenados los datos, las estructuras de datos en memoria son liberadas.

5.3.4. Inserción y eliminación silenciosa

Para cada operación fundamental existe una versión denominada como silenciosa porque no produce eventos. Estas versiones son útiles porque reproducen el mismo comportamiento inobservable en el exterior que tienen las operaciones de bases de datos. Estas operaciones se pueden aplicar, por ejemplo, durante la fase de inicialización de la base de datos en donde solamente resulta de interés su contenido.

5.3.5. Suscripción para recibir objetos de un tipo dado

La suscripción en colecciones de datos sigue la misma mecánica que la suscripción con mensajes. Las operaciones `inserted()` y `deleted()` realizan la suscripción del *thread* en ejecución por la inserción o eliminación de un objeto del mismo tipo que aquel indicado en el parámetro de estas operaciones.

5.3.6. Aceptación síncrona

La aceptación síncrona es como la definida para los conectores de mensajería.

5.3.7. Aceptación selectiva

La aceptación selectiva es como la definida para los conectores de mensajería.

5.3.8. Aceptación transaccional

La aceptación transaccional es como la definida para los conectores de mensajería excepto que es dependiente de las características transaccionales de la base de datos. Puesto que la base de datos se usa como almacenamiento estable de los datos, es importante definir mecanismos coherentes entre las características transaccionales de ADM y del manejador de base de datos elegido. Este es un problema que merece estudios adicionales y que no se abordaron en este trabajo. La solución que aquí se usó es la más simple: aplicar la operación `commit` (`rollback`) en la base de datos cuando se alcanza la operación `accept()` (`reject()`) en ADM.

5.3.9. Aceptación de eventos compuestos

La aceptación de eventos compuestos en colecciones de datos es similar a aquella en la comunicación por intercambio de mensajes. La principal diferencia radica en la presencia de operaciones de inserción y de eliminación, antagónicas entre sí. Se puede utilizar o definir álgebras de eventos para optimizar algunas operaciones (*efecto neto*) como la idempotencia de la eliminación de un mismo registro o la inserción seguida de la eliminación de un mismo registro, etc. Sin embargo, la versión actual del Administrador de Eventos de ADM no considera dichas optimizaciones.

5.3.10. Interacción tipo cliente-servidor

Al igual que el conector de mensajería, este conector adopta el paradigma de interacción cliente-servidor como principio de diseño de la arquitectura de las aplicaciones distribuidas. Cada conector de colecciones de datos se puede ver como un servidor que recibe peticiones para crear, insertar, eliminar o buscar registros en una base de datos relacional. El conector usa la tecnología disponible de los puentes JDBC-ODBC para garantizar independencia e interoperabilidad entre diferentes plataformas y lenguajes.

5.3.11. Estructuras de datos

Las estructuras de datos son muy similares a las ya presentadas debido a que usan el mismo patrón de diseño de publicación-suscripción.

Estructuras de control

Las estructuras de control permiten obtener procesos compuestos mediante construcciones estructuradas de alto nivel como las provistas por la clase `Activity` analizada antes.

- **CollectionConnector.** Representa un espacio de datos que se introducen, se eliminan o se recuperan sin eliminarse de acuerdo al conjunto de operaciones básicas que se describen a continuación.
 - Constructor `CollectionConnector(String url)`. Crea un nuevo conector que guarda conexión con el conector remoto situado en la dirección dada por el `url`. Los datos insertados, eliminados o recuperados en el conector local serán aplicados en el conector remoto. De esta manera, toda operación local realizará la operación correspondiente en el conector remoto.
 - Método `silentInsert(Head, Body)`. Inserta el registro dado por `Body` en la tabla sin producir evento alguno. Esta operación utiliza directamente la operación `insert` provista por el sistema administrador de la base de datos. El encabezado `Head` contiene el identificador del registro que puede usarse como llave primaria.

- Método `silentInsert(Body)`. Inserta el registro dado por `Body` en la tabla sin producir evento alguno y sin utilizar llave primaria.
- Método `insert(Head, Body)`. Inserta el registro dado por `Body` en la tabla produciendo un evento de inserción. Esta operación utiliza `silentInsert()`.
- Método `insert(Body)`. Inserta el registro dado por `Body` en la tabla sin utilizar llave primaria pero produciendo un evento de inserción.
- Método `silentDelete(Head, Body)`. Elimina el registro dado por `Body` en la tabla sin producir evento alguno. Esta operación utiliza directamente la operación `delete` provista por el sistema administrador de la base de datos. El encabezado contiene el identificador del registro que puede usarse como llave primaria.
- Método `silentDelete(Body)`. Elimina el registro dado por `Body` en la tabla sin producir evento alguno y sin utilizar llave primaria.
- Método `delete(Head, Body)`. Elimina el registro dado por `Body` en la tabla produciendo un evento de eliminación. Esta operación utiliza `silentDelete()`.
- Método `insert(Body)`. Elimina el registro dado por `Body` en la tabla sin utilizar llave primaria pero produciendo un evento de eliminación.
- Método `select(Head, Body)`. Recupera todos aquellos registros cuyo tipo sea el mismo que el que posee el registro dado `Body` y cuya llave primaria esté dada en el encabezado `Head`.
- Método `select(Body)`. Recupera todos aquellos registros cuyo tipo sea el mismo que el que posee el registro dado `Body`.

5.4. Detalles de implementación

Como se mencionó en el capítulo 4, una característica de nuestro sistema es que se usaron servicios Web para implementarlo. Un servicio Web, es una aplicación servidor que implementa procedimientos que están disponibles para que los clientes hagan uso de ellos. Los servicios están disponibles en un contenedor del lado del servidor, el contenedor puede ser un contenedor de servlets tal como Tomcat o una Plataforma Java 2 (p.ej el contenedor Enterprise Edition server-J2EE).

Un servicio Web puede hacerse disponible por sí mismo para clientes potenciales, describiéndose en un documento WSDL (Web Services Description Language), para mas detalles puede ver el apéndice ?? . Un consumidor (Cliente Web) puede usar el documento WSDL para saber que ofrece el servicio y como acceder a el mismo.

5.4.1. Tecnología

Los servicios Web son, como su nombre lo indica, servicios que se ofrecen vía Web. En un escenario típico de servicios Web, una aplicación de negocios envía una petición de un servicio a un URL específico usando el protocolo SOAP sobre HTTP. El servicio recibe la petición, la procesa y devuelve una respuesta. Un ejemplo clásico de servicio Web es un servicio de un almacén, en el cual se pide el precio actual de un artículo. En este caso, la petición y la respuesta son partes del la misma llamada a un método. Los servicios Web y los clientes de servicios Web son típicamente empresas (*businesses*), lo que hace que los servicios Web sean predominantemente transacciones business-to-business (B-to-B). Una empresa puede ser proveedor de servicios Web y también consumidor de otros servicios Web [].

Los servicios de Web dependen de la capacidad de las partes para comunicarse entre si, incluso aunque usen sistemas de información diferentes. XML (Extensible Markup Language), es una tecnología clave para cubrir esta necesidad debido a que hace que los datos sean portables. Como se mencionó en el capítulo 2, las empresas han descubierto las ventajas de usar XML para almacenar y compartir datos dentro de la empresa y con otras empresas. XML se ha convertido en un pilar para la computación relacionada con el Web.

Los servicios Web también dependen de la capacidad de las empresas para comunicarse entre si usando diferentes plataformas. Este requerimiento hace de la plataforma Java una buena elección para desarrollar servicios Web. La elección es mas atractiva con las nuevas APIs de Java para XML. Debido a que XML y la plataforma Java trabajan muy bien juntos, juegan un papel central en los servicios Web. Las APIs de Java para XML permiten crear aplicaciones Web completamente en el lenguaje de programación Java y se describen a continuación.

- Java API for XML Processing (JAXP) - procesa documentos XML usando varios analizadores gramaticales
- Java API for XML-based RPC (JAX-RPC) - envía llamadas a métodos SOAP hacia lugares remotos sobre Internet y recibe los resultados. (basada en SOAP)
- Java API for XML Messaging (JAXM) - envía mensajes SOAP sobre Internet en una forma estándar (basada en -SOAP)
- Java API for XML Registries (JAXR) - provee un manera estándar para acceder a registros de negocios e información compartida.

Las APIs mencionadas están disponibles en el Java Web Services Developer Pack (Java WSDP) 1.1, que es un paquete que contiene tecnologías para simplificar la construcción de servicios Web usando la plataforma Java. Las tecnologías que contiene la versión la versión que usamos son: JAXB (Java API for XML Binding), JAXM, SAAJ (Soap with Attachments API for Java), JAXP, JAXR, JAX-RPC, JSTL (JavaServer Pages Standard Tag Library), Tomcat (Contenedor de Java servlet y JavaServer Pages).

El servidor de servicios Web que se usó es el IBM WebSphere SDK for Web Services V5.0.1 (WSDK). El WSDK es un conjunto de herramientas que ayuda a escribir servicios Web usando la plataforma Java. WSDK incluye su propio registro UDDI. UDDI (Universal Description, Discovery and Integration) es una plataforma independiente del *framework* para describir servicios, encontrar negocios, e integrar servicios de negocios usando Internet. UDDI es un directorio para almacenar información a cerca de servicios Web. Tales servicios deben estar descritos con WSDL. WSDL se comunica vía SOAP y está construido en la plataforma Microsoft .NET. UDDI usa estándares de W3C e IETF como: XML, HTTP, y protocolos DNS.

Capítulo 6

Conclusiones y trabajo futuro

En estas tesis se hicieron extensiones al lenguaje ADM, mismo que hasta antes de este trabajo implementaba las características de reactividad y deducción en bases de documentos XML. La reactividad se refiere a la capacidad de detectar modificaciones que ocurren en las bases de documentos y reaccionar a ellas modificando la información del repositorio de documentos. La deducción permite hacer consultas sobre información que no se encuentra de forma explícita en la BDX, pero que se puede inferir a partir de información explícita y un conjunto de reglas sobre tal información.

Las extensiones que se hicieron al lenguaje consisten en un conjunto de primitivas que agregan funcionalidad a ADM. Las primitivas que se agregaron son: **send**, **received**, **insert**, **delete** e **inserted**. En este trabajo se extendió el dominio de aplicaciones de ADM ya que agregando únicamente estas 5 primitivas se incorporaron dos tecnologías más al lenguaje: los servicios Web y las Bases de datos relacionales.

Las primitivas **send** y **received** se relacionan directamente con los servicios Web, pero esto no significa que esa sea la única utilidad que tienen. La primitiva **send** hace posible enviar mensajes XML a un URL específico, la contraparte de esta es la primitiva **received**, que permite recibir información que fue enviada desde un URL específico. La vinculación de los URLs con el lenguaje se hizo por medio de **conectores**, mismos que se proponen en [Agu00].

Los servicios Web son un conjunto de aplicaciones con capacidad para interoperar en la Web. Los servicios Web se ofrecen como procedimientos remotos que los usuarios solicitan mediante llamadas a través de la Web. Por lo antes mencionado, la incorporación de esta tecnología al lenguaje ADM proporciona grandes ventajas como son:

- El ámbito de aplicación de las reglas ECA con la incorporación de servicios Web se extiende a la Web. Ahora mediante este lenguaje es posible la interacción en la Web de una forma simple. Para hacer peticiones de servicios Web se usa la primitiva **send**, esta primitiva puede contener cualquier documento XML, mismo que será enviado al URL que se indique en la primitiva y que corresponde al URL en donde se ofrece el servicio Web. En el contenido de ese documento se especifica el nombre de la operación que se solicita del servicio y los parámetros necesarios para ejecutar la operación. La primitiva **received** permite recibir documentos XML que se han enviado como respuestas de servicios Web (o de cualquier URL).
- Usando reglas ECA se puede invocar servicios Web de forma automática. Esto se logra cuando en una regla ECA se especifica en la parte de la acción que se envíe un mensaje a un URL solicitando una operación de un servicio Web.

- La respuesta de un servicio Web puede activar a otro servicio. Se pueden crear reglas para que cuando se reciba la respuesta de un servicio Web se analice su contenido y en base a ese contenido se activen otras reglas.

Por otro lado, las primitivas `insert`, `delete`, e `inserted` corresponden al manejo de bases de datos. Estas primitivas existían para manipular BDX, por lo que se hicieron las adecuaciones necesarias para que soporten también el uso de bases de datos relacionales. La modificación principal que se hizo a estas primitivas consiste también en el uso de `conectores`. En esta tesis, los conectores se implementaron de tal forma que permiten diferenciar entre bases de datos y bases de documentos.

El resultado de esta tesis es un lenguaje que combina diversas tecnologías para el manejo de información, estas son: reactividad, deducción, bases de documentos, bases de datos relacionales y servicios Web.

Un lenguaje con las capacidades actuales de ADM es una herramienta poderosa para la implementación de aplicaciones que automatizan tareas sobre la Web, o que requieren deducción sobre Web. Además como ADM trabaja con SOAP como protocolo de intercambio de información y con XML, esto hace a ADM portable.

Para validar las extensiones que se hicieron a ADM, se implementó un caso de estudio de un servidor proxy de páginas Web, mejor conocido como proxy http o proxy Web. La función principal de un proxy http consiste en almacenar páginas Web que solicitan los clientes a los servidores de Internet, para que posteriormente cuando los clientes pidan estas páginas, sea el proxy http el que las proporcione. Debido a que el proxy minimiza el número de accesos remotos, esto reduce el tráfico en la red. Los proxy http también disminuyen el tiempo para obtener los datos y permiten que muchos usuarios tengan acceso a Internet usando sólo una conexión.

En el caso de estudio que se implementó en esta tesis, se manejan cuatro participantes que son: un cliente, un servidor interno, un servidor externo y un proxy-http. El cliente hace peticiones de páginas Web, estas peticiones las recibe y atiende el proxy. El servidor interno contiene las páginas que los clientes ya ha pedido alguna vez, es decir, funciona como la caché del proxy. El servidor externo funciona como cualquier servidor de páginas Web que se encuentre sobre Internet.

El funcionamiento de los cuatro participantes se modeló mediante un conjunto de reglas ECA para cada uno. Cada participante puede estar en una máquina por separado. La implementación de este caso de estudio se hizo con el lenguaje Java, Tomcat, y el servidor de servicios Web IBM WebSphere.

Aunque ADM es un lenguaje que actualmente tiene muchas capacidades, aún es posible mejorarlo, por ejemplo se puede desarrollar un entorno gráfico que permita crear, editar y monitorear la ejecución de reglas ECA usando UML. Esto facilitaría al usuario la creación de las reglas ECA ya que el usuario sólo tendría que crear los diagramas correspondientes UML y el sistema generaría el código de forma automática. Además, el usuario podría ver reflejado en pantalla lo que va sucediendo durante la ejecución de las reglas.

El mecanismo de resolución de conflictos de un sistema que maneja reglas ECA, sirve para escoger una regla de entre un conjunto de reglas que se activan con el mismo evento (a este conjunto de reglas se le denomina *conjunto conflicto*). El mecanismo de resolución de conflictos que tiene ADM a la fecha, es *injusto* (unfair), ya que siempre escoge la primera regla del conjunto conflicto. Como se mencionó en el capítulo 1 existen una gran variedad de algoritmos para elegir una regla. Otra mejora a ADM consiste en implementar algún algoritmo de resolución de conflictos que no sea injusto.

Otra mejora que se puede hacer a ADM es ejecutar las reglas en paralelo. Las reglas ECA que se describen usando este lenguaje, se podrían especificar de forma tal que un mecanismo haga

la ejecución de varias reglas al mismo tiempo siempre y cuando estas reglas no produzcan un conflicto. Con un mecanismo que agregue esta capacidad al lenguaje, este lenguaje sería mucho más poderoso.

A la fecha, ADM emplea la *granularidad de regla* [BCP01] orientada a nodos. Es decir, considera cada evento como una instancia separada y la variable en la parte del evento se asocia con un único evento. Se puede modificar ADM para que la granularidad de regla sea orientada a conjunto.

Finalmente destacamos que el lenguaje ADM con las capacidades que cuenta actualmente permite la creación de diversas aplicaciones que son de gran utilidad, por ejemplo: alertadores, personalización, clasificadores, mantenimiento de vistas y workflows totalmente automatizados sobre Internet.

Los alertadores se usan para enviar información a usuarios interesados en ella cuando la información se acaba de ingresar al repositorio de documentos.

La personalización, o entrega de sitios Web uno a uno, ha llamado la atención debido a su gran fuerte. La personalización normalmente es precedida por la adquisición de información del perfil del usuario, y se usa esta información para cambiar el contenido de la página Web que se está presentando. Por ejemplo, poniendo *banners* o presentando propaganda que corresponde al interés del usuario.

Los clasificadores son un ejemplo clásico de aplicación de reglas ECA, tienen diversas utilidades, por ejemplo se pueden usar para checar mensajes de entrada de correo electrónico, eliminar algunos de ellos o ponerlos en diferentes carpetas para leerlos mas tarde.

El mantenimiento de vistas es una aplicación clásica de las reglas activas. Una vista es simplemente un documento derivado, cuyo contenido puede ser ensamblado consultando otros documentos. El servicio de mantenimiento de vistas se puede usar para mantener la vista de acuerdo a los documentos originales cuando estos son modificados.

Los workflows normalmente separan las actividades de trabajo en tareas bien definidas, papeles, reglas, y los procedimientos que regulen la mayor parte del trabajo en la fabricación y la oficina [GHS95]. Un ejemplo son los programas de computadora que realizan tareas y forzan el cumplimiento de reglas que fueron previamente implementadas por humanos. Con ADM los workflows sobre Internet puede ser totalmente automatizados.

Bibliografía

- [AFJ98] Martin Arlitt, Rich Friedrich, and Tai Jin. Performance evaluation of Web proxy cache replacement policies. *Lecture Notes in Computer Science*, 1469:193+, 1998.
- [Agu00] José Oscar Olmedo Aguirre. *Design and Construction of a Coordination Language with Applications to Hypermedia*. PhD thesis, Faculty of Engineering and Applied Science, Department of Electronics and Computer Science, University of Southampton, December 2000.
- [BCP01] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Active rules for xml: A new paradigm for e-services. *VLDB J.*, 10(1):39–47, 2001.
- [BCP02] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Pushing reactive services to xml repositories using active rules. *Computer Networks*, 39(5):645–660, 2002.
- [BLC95] T. Berners-Lee and D. Connolly. RFC 1866: Hypertext Markup Language — 2.0, November 1995. Status: PROPOSED STANDARD.
- [BPW01] J. Bailey, A. Poulouvasilis, and P. Wood. Analysis and optimisation for eventcondition-action rules on xml, 2001.
- [CI97] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, 1997.
- [Esc04] Karina Escobar Vázquez. Atribución de significado a documentos xml con logcin-xml. Master’s thesis, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional - Unidad Zacatenco, D.F, México, April 2004.
- [GHS95] Dimitrios Georgakopoulos, Mark F. Hornick, and Amit P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.
- [HW93] E.Ñ. Hanson and J. Widom. An Overview of Production Rules in Database Systems. *The Knowledge Engineering Review*, 8(2):121–143, June 1993.
- [IC97] Arun Iyengar and Jim Challenger. Improving web server performance by caching dynamic data. In *USENIX Symposium on Internet Technologies and Systems*, 1997.
- [Kil] Pekka Kilpeläinen. Sgml & xml content models.
- [SM] Edited By Sperberg-Mcqueen. Guidelines for electronic text encoding and interchange.
- [SR01] L. Seligman and A. Rosenthal. The impact of xml on databases and data sharing, 2001.
- [vdG97] M. van der Graaf. A specification of box to html in asf+sdf, 1997.

Apéndice A

Introducción del lenguaje

El propósito de este capítulo es proporcionar un panorama general del lenguaje ADM y mostrar el estilo de programación lógica y el manejo de eventos que se pueden conseguir. Para tal efecto se desarrolló una aplicación que permite demostrar las capacidades activas y deductivas del sistema. La aplicación corresponde al conocido problema de parentesco familiar que consiste en definir reglas que permitan deducir cuál es el parentesco que relaciona a dos individuos. Esta aplicación permite mostrar las dos características más sobresalientes del lenguaje tales como:

- *La capacidad para recuperar información que relaciona a dos o más documentos de acuerdo con un conjunto preciso de condiciones lógicas.* Este enfoque contrasta con el seguido en la mayoría de los motores de búsqueda tradicionales que agrupan documentos por la similitud de la frecuencia relativa de palabras clave en vez de relacionarlos por las condiciones lógicas que cumple la estructura y el contenido de dichos documentos.
- *La capacidad de reaccionar ante eventos.* Estas reacciones se reflejan en que el contenido de la colección de documentos se ve afectado mediante la inserción de nuevos documentos, la eliminación de algunos ya existentes o el intercambio de documentos mediante los servicios de mensajería de los servicios Web. Cuando sucede un evento en una BDX, el administrador de reglas selecciona aquella que sea apropiada de acuerdo al tipo de evento detectado y los parámetros del mismo. Si los parámetros del evento verifican la condición de la regla de acuerdo con el contenido de la BDX, entonces se ejecutan las acciones que pueden consistir en insertar, borrar o enviar un documento.

El lenguaje ADM está formado por elementos XML, términos XML, procedimientos lógicos y consultas. Mientras que la parte activa del lenguaje se forma con eventos, acciones y reglas evento-condición y acción. A continuación se describen los componentes deductivo y activo de ADM.

A.1. Reglas deductivas en ADM

El enfoque que se adopta en este trabajo para proporcionar las capacidades deductivas de ADM consiste en extender a los elementos XML con variables lógicas y procedimientos lógicos.

A.1.1. Elementos XML

Los elementos XML constan del nombre del elemento y una lista posiblemente vacía de atributos (pares nombre y valor). Los atributos están ordenados lexicográficamente con base en su

```

<persona>
  <nombre sexo = "femenino">Anahi</nombre>
  <edad>24</edad>
</persona>

```

Figura A.1: Elemento XML

```

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

```

Figura A.2: Término XML

nombre. 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 delimitados por comillas. La diferencia se muestra en la Figura A.1, en donde se puede apreciar que el elemento `persona` está compuesto a su vez por los elementos `nombre` y `edad`, cuyos valores respectivos son `Anahi` y `24`, ambos de tipo texto. El elemento `nombre` posee además una lista de atributos, dicha lista consiste únicamente del atributo `sexo` que tiene el valor `femenino` y es de tipo cadena de caracteres.

La estructura de las listas de atributos y de elementos es muy similar a la estructura de los términos en lógica formal, lo que permite descomponer un documento en sus piezas básicas de información usando los métodos usuales de programación en lógica. En la versión actual de ADM, no se dispone de facilidades para denotar elementos y atributos por expresiones XPath[REF]. De contar con dichas facilidades se conseguiría simplificar considerablemente el tamaño de las expresiones usadas para extraer la información de elementos y atributos. Sin embargo, desde un punto de vista formal, se puede alcanzar una expresividad comparable a la de XPath con las construcciones de ADM que aquí se presentan.

A.1.2. Términos XML

Desde el punto de vista de la programación lógica, los elementos XML corresponden a términos sin variables. ADM extiende los elementos XML con la noción de variable lógica para introducir términos XML. La importancia de las variables lógicas radica en que permiten definir las condiciones que debe cumplir la información deducida por el sistema. En ADM, las variables lógicas son nombres simbólicos que comienzan con `$` (signo de pesos). Las variables anónimas están permitidas y se designan por `$` (signo de pesos seguido por una subraya).

En la Figura A.2 se escribe el elemento `persona` como un término con variables. Las variables `$S`, `$N` de tipo texto y `$E` de tipo elemento se introducen para establecer condiciones lógicas que debe cumplir el contenido de los documentos con quienes se comparen. Por ejemplo, si los términos XML presentados en las las Figuras A.1 y A.2 se consideran sintácticamente iguales, entonces las variables `$S` y `$N` deben ser iguales a los valores `femenino` y `Anahi`, respectivamente, mientras que la variable `$E` debe ser igual al elemento `<edad>24</edad>`.

A.1.3. Procedimientos lógicos

Los procedimientos lógicos representan los axiomas que describen formalmente las características atemporales de un sistema, estableciendo las restricciones lógicas que debe cumplir la información contenida en la colección de documentos. Para teorías de primer orden que usan el lenguaje de cláusulas de Horn, los axiomas se pueden interpretar como procedimientos. Así, el cuerpo del procedimiento describe la secuencia de acciones cuya ejecución conduce a inferir aquella información que satisface las restricciones establecidas por el procedimiento y por los axiomas del sistema.

Sintácticamente, un procedimiento lógico consiste de una secuencia de invocaciones a otros procedimientos lógicos. En ADM se pueden clasificar en: procedimientos primitivos, predicados básicos y predicados genéricos definidos por el programador. En relación al alcance de los nombres, se puede decir que, en general, el nombre de un procedimiento tiene ámbito global mientras que el nombre de una variable tiene ámbito local al cuerpo del procedimiento.

Entre las características más sobresalientes del modelo de programación lógica de ADM se encuentra su capacidad de buscar y extraer información contenida en 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]. Como se discutirá más adelante, en ADM es posible generar el contenido de nuevos documentos a partir de la información extraída de la colección usando métodos de programación lógica y de satisfacción de restricciones.

A continuación se describen con más detalle los diferentes tipos de procedimientos que se pueden definir en ADM.

Predicados primitivos

Los predicados primitivos son aquellos predefinidos por el sistema. En muchas ocasiones, los predicados primitivos amplían considerablemente el poder del modelo de programación lógica. Sin embargo,, 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 pueden agrupar en las siguientes categorías:

- Unificación y comparación de términos.
- Conjuntos de soluciones.
- Meta-predicados.
- Modificación de programas.
- Aritmética.
- Lectura/Escritura de términos.

En ADM los predicados primitivos se invocan siguiendo un estilo parecido al usado en los lenguajes imperativos como SmallTalk [REF]. Por ejemplo, para asignar el resultado de la evaluación de una expresión aritmética a una variable, el predicado primitivo `assign` se usa de la siguiente forma:

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

```

<assert>
  <hijo - de>
    <padre>Leopoldo</padre>
    <hijo>Ivan</hijo>
  </hijo - de>
</assert>

```

Figura A.3: Ejemplo del meta-predicado `assert`

```

<padre - de>
  <padre>Jaime</padre>
  <hijo>Leopoldo</hijo>
</padre - de>

```

Figura A.4: Instancia de la relación `padre-de`

en donde, los nombres de los atributos `variable` y `expression` identifican el papel que desempeñan los parámetros en la invocación del predicado primitivo `assign`. Este ejemplo muestra como se asigna a la variable `$Y` la suma del valor vinculado a la variable `$X` y 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 `assign` lo cual establece que su segundo parámetro debe ser de tipo numérico (similar al predicado `is/2` del Prolog estandard). En general, cada procedimiento primitivo establece el tipo de las expresiones que involucran, de modo que éstas sean de tipos compatibles.

Entre los predicados primitivos más importantes de `ADMse` encuentran los meta-predicados `assert` y `retract` definidos para modificar el contenido de los programas que han sido instalados en la memoria de trabajo del sistema. El predicado `assert` introduce un procedimiento (básico o genérico) a la colección sin considerar sus posibles repeticiones, mientras que el predicado `retract` elimina un predicado de la colección siempre que exista uno que concuerde con su estructura y contenido. Por ejemplo, para modificar el conocimiento que se tiene de las relaciones de parentesco, se usa el predicado `assert` como se indica en la Figura A.3 de modo que refleje el hecho de que `Leopoldo` tiene un nuevo hijo `Ivan`.

Al ejecutar el predicado que aparece en la Figura A.3, se introducirá un procedimiento a la colección que forma el programa. Si este meta-predicado se ejecutara, la colección de documentos instalados en la memoria de trabajo incluiría al nuevo documento que se muestra en la Figura A.4.

En esta figura se describe una instancia de la relación `padre-de`, en donde `Jaime` y `Leopoldo` juegan los papeles de padre e hijo, respectivamente. Para determinar cuál es el nombre del padre de `Leopoldo`, podemos usar la variable `$P` como se muestra en la Figura A.5. Al ejecutar tal procedimiento obtenemos la solución `$P = "Jaime"`.

En `ADM` se pueden asignar a las variables lógicas no sólo elementos simples sino también estructurados. Por ejemplo, el predicado primitivo `equal` usa el bien conocido procedimiento de unificación de términos lógicos para decidir sobre la igualdad sintáctica de términos. Así, la invocación: `<equal this = "$T" to = "<hijo>Leopoldo</hijo>" />` vincula a la variable `$T` con el término `<hijo>Leopoldo</hijo>`.

La expresión `<hijo>Leopoldo</hijo>` es una forma alternativa de escribir el elemento `<hijo>Leopoldo</hijo>` en donde las referencias `<` y `>` evitan usar los caracteres

```

<rule>
  <do>
    <call>
      <padre - de>
        <padre>$P</padre>
        <hijo>Leopoldo</hijo>
      </padre - de>
    </call>
  </do>
</rule>

```

Figura A.5: Invocación al procedimiento de la relación padre-de

```

<rule>
  <on>
    <call>
      <padre - de>
        <padre>Jaime</padre>
        <hijo>Leopoldo</hijo>
      </padre - de>
    </call>
  </on>
</do> </do>
</rule>

```

Figura A.6: Representación de la instancia padre-de como procedimiento básico

especiales ‘<’ y ‘>’. ADM interpreta correctamente este fragmento de texto como un elemento cuando aparece en el contexto apropiado. Aunque este mecanismo permite usar elementos XML en el paso de parámetros, su uso debe limitarse ya que la escritura de múltiples elementos anidados conduce a errores de sintaxis difíciles de identificar.

Procedimientos Básicos

El propósito del procedimiento básico es representar la información factual contenida en una colección de documentos así como recuperar dicha información mediante consultas. El procedimiento básico es una regla generalmente anónima que establece como un hecho la validez del procedimiento que aparece en el encabezado de la regla (en el elemento `on`).

La correspondencia entre documentos y procedimientos básicos se ejemplifica en la Figura A.6. La figura muestra la representación en forma de procedimiento básico del documento dado en la Figura A.4. Así, en general, para cada documento de la colección, existe un procedimiento básico que lo representa. Esta correspondencia permite utilizar uniformemente el modelo computacional basado en la interpretación procedural de las cláusulas de Horn [Loyd].

Una consulta a un procedimiento básico corresponde con la invocación de dicho procedimiento la cual generalmente incluye variables lógicas. Las invocaciones de procedimientos básicos pueden inclusive compartir variables para formar consultas más complejas. Por ejemplo, para saber si

```

<rule>
  <do>
    <call>
      <padre - de>
        <padre>$P</padre>
        <hijo>Leopoldo</hijo>
      </padre - de>
    </call>
    <call>
      <esposo - de>
        <marido>$P</marido>
        <mujer>Maria</mujer>
      </esposo - de>
    </call>
  </do>
</rule>

```

Figura A.7: Procedimientos generadores con variables compartidas

María es madre (política) de Leopoldo, podemos formular una consulta como la que se muestra en la Figura A.7. Esta consulta 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 Leopoldo y esposo de María. El concepto de variable lógica es fundamental para establecer relaciones entre los contenidos de dos o más documentos e incluir restricciones sobre ellos.

Al comparar la estructura de los procedimientos básicos con la estructura de las consultas se puede observar que son complementarias entre sí, es decir, mientras que en la primera solamente se define el encabezado de la regla (etiqueta `<on>`), en la última se define solamente su cuerpo (etiqueta `<do>`). En otras palabras, mientras que los procedimientos básicos se caracterizan por el mecanismo fundamental de invocación a un procedimiento, las consultas se caracterizan por el mecanismo de ejecución ordenada de invocaciones a predicados. Ambas construcciones se combinan en los así llamados procedimientos genéricos, los cuales permiten etiquetar consultas con encabezados, como se explica a continuación.

Procedimientos genéricos

Los procedimientos genéricos son procedimientos definidos por el programador que permiten introducir restricciones lógicas más complejas que aquellas que se pueden definir con procedimientos primitivos o básicos. Las restricciones se escriben en forma de una secuencia de invocaciones a predicados (primitivos, básicos o genéricos) que pueden pasar información de unos a otros mediante variables lógicas. Un conjunto de procedimientos lógicos con el mismo encabezado constituyen la definición del nombre del procedimiento. Cada uno de dichos procedimientos corresponde con definiciones alternativas que pueden utilizarse en una invocación. El orden en el que se seleccionan coincide con el orden con el que aparecen en el programa. El procedimiento genérico es una regla generalmente anónima que establece que la validez del procedimiento que aparece en el encabezado de la regla (elemento `on`) es consecuencia lógica de la validez de los procedimientos que aparecen en el cuerpo (elemento `do`). El cuerpo de un procedimiento se puede interpretar

```

<rule>
  <on>
    <call>
      <madre - de madre = "$Madre" hijo = "$Hijo" />
    </call>
  </on>
  <do>
    <call>
      <padre - de>
        <padre>$Padre</padre>
        <hijo>$Hijo</hijo>
      </padre - de>
    </call>
    <call>
      <esposo - de>
        <marido>$Padre</marido>
        <mujer>$Madre</mujer>
      </esposo - de>
    </call>
  </do>
</rule>

```

Figura A.8: Procedimiento que define la relación madre-de

entonces como una consulta, en donde los parámetros del procedimiento deben fijarse antes de formularla. Al igual que los procedimientos básicos, esta forma de representación utiliza el modelo computacional que se deriva de la interpretación procedural de las cláusulas de Horn [Lloyd].

Generalmente, los procedimientos se escriben siguiendo el estilo de programación conocido como *genera y prueba* (*generate and test*) que, como sugiere su nombre, consiste en generar soluciones para las variables mediante consultas a las bases de datos extensionales para seleccionar aquella solución que cumple con las restricciones indicadas en la prueba. Este estilo se basa en una técnica conocida como *retroceso* (*backtrack*) la cual utiliza cada una de las definiciones alternativas de un procedimiento para generar todas las soluciones posibles que se pueden obtener por combinación sistemática y exhaustiva de los valores que toman las variables en sus dominios respectivos.

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*.

Como se ha dicho, las invocaciones a procedimientos permiten relacionar documentos mediante variables lógicas. En la Figura A.8 se muestra la definición del procedimiento *madre-de* que usa este mecanismo para determinar si los nombres de dos personas están relacionadas a partir de las relaciones *padre-de* y *esposo-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.

```

<rule>
  <on>
    <madre - de madre = "$Madre" hijo = "$Hijo" />
  </on>
  <if>
    <padre - de>
      <padre>$Padre</padre>
      <hijo>$Hijo</hijo>
    </padre - de>
    <esposo - de>
      <marido>$Padre</marido>
      <mujer>$Madre</mujer>
    </esposo - de>
  </if>
  <do> </do>
</rule>

```

Figura A.9: Representación de la relación madre-de como regla ECA.

A.2. Reglas activas en ADM

En comparación con las reglas deductivas, las reglas activas remedian la falta de interactividad y de extensibilidad del servidor de aplicaciones. Las reglas activas definen el comportamiento reactivo que modifica el contenido de las bases de documentos en respuesta a los eventos de inserción, eliminación y recepción de documentos. La importancia de las reglas activas radica en que le ofrecen al programador mecanismos para definir e incorporar su propia infraestructura de control dentro del servidor de aplicaciones. Para introducir el modelo activo de programación en ADM, en este trabajo se ha desarrollado e incluido un administrador de reglas activas el cual es responsable de interactuar con otros componentes activos, clientes u otros servidores.

A.2.1. Estructura de las reglas activas

La Figura A.9 muestra el procedimiento genérico **madre-de** como regla activa. En la figura se puede apreciar la estructura de la regla, la cual posee tres elementos (**on-if-do**), en comparación con la estructura de los procedimientos genéricos que solamente posee dos (**on-do**). Desde el punto de vista sintáctico, la regla ECA se obtiene del procedimiento genérico al convertir el elemento **do** del procedimiento en el elemento **if**, removiendo al mismo tiempo las operaciones **call** de invocación. Desde el punto de vista semántico, la interpretación procedural permite identificar una secuencia de invocaciones a procedimientos lógicos como una consulta cuya sintaxis se ha simplificado para evitar la necesidad de incluir explícitamente las operaciones de invocación. De esta forma, para cada procedimiento genérico es posible escribir una regla activa que le corresponda. Al igual que en los procedimientos genéricos, la utilización del retroceso en la consulta asegura la generación de todas las soluciones posibles para las variables. En caso de que por este medio la consulta genere más de una solución, entonces para cada solución se ejecutan las acciones correspondientes indicadas por la regla después de substituir variables.

La Figura A.10 muestra los procedimientos lógicos **hermano-de** y **primo-de**, en tanto que la Figura A.11 muestra la regla activa **invitacion**. Los procedimientos definen las relaciones de

parentesco del mismo nombre en la forma usual. Observe que estos procedimientos están escritos como reglas activas con acción nula porque las reglas tienen una apariencia más simple y legible que los procedimientos. Por otra parte, la regla `invitacion` es una regla activa completa.

Selección de las reglas

La regla se selecciona cuando se recibe un nuevo documento como notificación de que un nuevo miembro de familia ha nacido. Se supone para este ejemplo que el contenido de la base extensional de documentos es el que se muestra en la Figura A.12. La Figura A.13 muestra el documento recibido por el módulo activo. El módulo activo de ADM notifica al sistema que ha ocurrido un evento de recepción de un mensaje cuya estructura es la misma que la del elemento `on` indicado en el evento de la regla. El módulo de recepción de mensajes envía el contenido al administrador de reglas de ADM para seleccionar la regla apropiada. El administrador mantiene una lista de reglas indexadas por el tipo de operación y de evento. De esta manera, la regla `invitacion` está indexada por la operación `receive` y por el evento `padre-de`. El administrador entonces extrae el contenido del mensaje y con él formula la consulta indicada en la condición de la regla. Si la consulta indicada en la condición no tiene solución, entonces la acción de la regla no se ejecuta. En otro caso, la regla se ejecuta como se explica a continuación.

Respuesta de la regla

Se obtiene una respuesta de la regla activa cuando se ejecuta su acción. Sin embargo, la acción solamente se ejecuta si la consulta dada por la condición tiene al menos una solución. En caso de existir varias soluciones, para cada una de ellas se ejecutarán las acciones correspondientes.

Las variables lógicas `$X` y `$Y` se usan para extraer las piezas de información que coincide su posición con el evento dado en la regla. Se aplica una sustitución a la regla (que incluye las vinculaciones `$X = "Tomas"` y `$Y = "Arturo"` si el documento insertado corresponde con el documento que se muestra en la Figura A.13) para obtener una instancia de la regla. Entonces el motor de inferencia deduce una solución `$Z = "David"` a partir de la consulta `<primo - de subject = "$Y" object = "$Z" />` y ejecuta la parte de acción de la regla insertando el documento que se muestra en la Figura A.14.

A.3. Resumen

El lenguaje ADM acopla las características de reactividad y deducción en un modelo uniforme de programación. En este capítulo se mostraron algunas características del lenguaje.

Las reglas deductivas consisten de elementos XML, términos XML, procedimientos y consultas. Los elementos XML consisten del nombre del elemento y una lista de atributos con sus respectivos valores (la lista puede estar vacía). Los términos son elementos XML que se extienden en ADM para crear términos con variables. Los procedimientos se componen de secuencias de acciones que obtienen la información que cumple las restricciones establecidas por los axiomas que describen las características del sistema. En ADM hay tres tipos de procedimientos: procedimientos primitivos, básicos y genéricos. Las consultas se usan para recuperar información que es consecuencia lógica del contenido de la BDX.

Por otro lado, las reglas activas del lenguaje proporcionan mayor flexibilidad al programador para ejercer cierto control en el servidor de aplicaciones. Los eventos que se emplean en ADM son: inserción, eliminación y recepción (fragmentos) de documentos XML. Los eventos son detectados por el administrador de reglas activas el cual selecciona la regla apropiada (si la hay) para realizar las acciones correspondientes en respuesta al evento detectado. Aunque la descripción de ADM

```

<rule name = "hermanode">
  <on>
    <hermano - de subject = "$X" object = "$Y">
  </on>
  <if>
    <notequal subject = "$X" object = "$Y" />
    <padre - de>
      <padre>$Z</padre>
      <hijo>$X</hijo>
    </padre - de>
    <padre - de>
      <padre>$Z</padre>
      <hijo>$Y</hijo>
    </padre - de>
  </if>
  <do> </do>
</rule>

```

```

<rule name = "primode">
  <on>
    <primo - de subject = "$X" object = "$Y">
  </on>
  <if>
    <padre - de>
      <padre>$Z1</padre>
      <hijo>$X</hijo>
    </padre - de>
    <padre - de>
      <padre>$Z2</padre>
      <hijo>$Y</hijo>
    </padre - de>
    <hermano - de subject = "$Z1" object = "$Z2" />
  </if>
  <do> </do>
</rule>

```

Figura A.10: Documento de las relaciones de una familia en una base de datos intensional


```

<rule name = "invitacion">
  <on>
    <receive>
      <padre - de>
        <padre>$X</padre>
        <hijo>$Y</hijo>
      </padre - de>
    </receive>
  </on>
  <if>
    <primo - de subject = "$Y" object = "$Z" />
  </if>
  <do>
    <send>
      <invitacion>
        <anfitrion>$Y</anfitrion>
        <invitado>$Z</invitado>
      </invitacion>
    </send>
  </do>
</rule>

```

Figura A.11: Regla activa completa

```

<padre - de>
  <padre>Benito</padre>
  <hijo>Juan</hijo>
</padre - de>

```

```

<padre - de>
  <padre>Benito</padre>
  <hijo>Tomas</hijo>
</padre - de>

```

```

<padre - de>
  <padre>Juan</padre>
  <hijo>David</hijo>
</padre - de>

```

Figura A.12: Contenido de la BDX extensional de las relaciones de una familia

```

<padre – de>
  <padre>Tomas</padre>
  <hijo>Arturo</hijo>
</padre – de>

```

Figura A.13: Documento XML que al ser recibido activa la regla `invitacion`

```

<invitacion>
  <anfitrión>Tomas</anfitrión>
  <invitado>David</invitado>
</invitacion>

```

Figura A.14: Documento producido como respuesta a la activación de la regla `invitacion`

dada en este capítulo proporciona los elementos para entender las construcciones del lenguaje, es necesario ofrecer una descripción más formal que provea una base sólida para el análisis de las propiedades de los programas y sirva de guía para realizar alguna implementación del lenguaje. En el siguiente capítulo se presentará el modelo formal de las reglas activas.

En la Figura A.15 se muestran algunos ejemplos de las relaciones que se pueden definir para el problema de las relaciones de parentesco. La relación `datos – personales` se introdujo 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 se requiera.

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

Cuando hay cambios en una colección de documentos XML ya sea por inserción o eliminación de un documento, ocurre un evento E , y entonces seleccionan una o más reglas. Una vez que la regla se selecciona, se verifica la condición C y si ésta se satisface mediante el contenido de la BDX, entonces se ejecutan las acciones de borrar el documento B , insertar el documento A , no necesariamente en ese orden. Se puede definir el ordenamiento de acciones mediante la composición de regla R ya sea en forma secuencial o paralela.

```
<padre - de>
  <padre>Jaime</padre>
  <hijo>Leopoldo</hijo>
</padre - de>

<esposo - de>
  <marido>Jaime</marido>
  <mujer>Maria</mujer>
</esposo - de>

<datos - personales>
  <nombre>Leopoldo</nombre>
  <edad>20</edad>
  <estado - civil>soltero</estado - civil>
  <ocupacion>comerciante</ocupacion>
  <sexo>masculino</sexo>
</datos - personales>
```

Figura A.15: Documentos de una DBX

Apéndice B

Conceptos básicos y tecnologías relacionadas con XML

B.1. Lenguajes de marcado

B.1.1. SGML

SGML (Standard Generalized Markup Language), según [vdG97] SGML «Define un marco de trabajo en el cual pueden definirse lenguajes de marcado para diferentes propósitos», debido a esta característica, se dice que es un metalenguaje. Mediante este lenguaje se han definido HTML, XML y XHTML

Una contribución importante de SGML es que introduce la noción de *tipo de documento* y de ahí una Definición de Tipo de Documento (DTD). Las DTD permiten distinguir lenguajes de marcado particulares y contienen la especificación formal de la sintaxis.

B.1.2. HTML

HTML (Hyper Text Markup Language) es un lenguaje de marcado simple que sirve para crear documentos de hipertexto independientes de la plataforma [BLC95]. Este lenguaje se deriva de SGML. HTML está compuesto por un conjunto predefinido de etiquetas que tienen como función indicar al navegador la forma en que debe desplegar los datos. Por ello, se dice que los objetivos de HTML son: dar formato a los datos y mostrarlos.

HTML es el lenguaje en el que se escriben y publican las llamadas páginas Web. La Web está organizada y dirigida por el consorcio de la Web - W3C¹.

B.1.3. XML

XML al igual que SGML es un metalenguaje pues usando XML se han creado nuevos lenguajes como WSDL (Web Services Description Language). WAP (Wireless Application Protocol) y WML (Wireless Markup Language). WML se usa en aplicaciones de Internet para dispositivos inalámbricos como teléfonos móviles. WAP es un protocolo que ofrece intercambio seguro de información por Internet. WSDL es un lenguaje que sirve para describir servicios Web.

¹El Consorcio World Wide Web se creó en octubre de 1994 para guiar la Web a su máximo potencial mediante el desarrollo de protocolos comunes que promuevan su evolución y aseguren su interoperabilidad. El W3C tiene cerca de 400 Organizaciones Miembro en todo el mundo y ha ganado reconocimiento internacional por sus contribuciones al crecimiento de la Web. En diez años, W3C ha desarrollado más de ochenta especificaciones técnicas para la infraestructura del Web.

La diferencia principal entre SGML y XML es que éste último no requiere que una DTD acompañe el documento. XML sólo requiere que los documentos estén bien formados (sean gramaticalmente correctos). Al decir bien formados, nos referimos a que el principio y el final de todos los elementos del documento deben estar marcados correctamente, anidando los pares de las etiquetas de marcado que correspondan [Kil].

Existen muchas diferencias entre XML y HTML por ejemplo: XML permite al usuario definir sus propias etiquetas, mientras que el autor de documentos HTML sólo puede usar las etiquetas que están definidas en el estándar de HTML. Otra diferencia es que XML permite al autor definir su propia estructura de documentos, mientras que la estructura de documentos de HTML también está predefinida. Sin embargo la diferencia principal entre XML y HTML es que el primero fue diseñado para describir y transportar datos, mientras que el segundo fue diseñado para darles formato y mostrarlos.

Dado que la funcionalidad de XML y HTML es diferente, se puede decir que XML no es un reemplazo para HTML sino un complemento.

En la actualidad, muchas intranets usan HTML como el formato predominante para publicar información, pero las necesidades de los usuarios crecen, por lo que se requiere que los lenguajes ofrezcan mayor flexibilidad. HTML es un lenguaje orientado a la presentación, es decir, se enfoca en la interfaz computadora-humano. Sin embargo se necesita un lenguaje orientado al contenido para que se centre en aplicaciones de computadora a computadora (como la transferencia de la información entre bases de datos), por lo que se creó XML. En otras palabras, HTML que está dotado para construir sistemas en los cuales los usuarios interpretan los datos, pero si se requiere que las aplicaciones sean quienes interpreten los datos entonces se debe usar XML.

B.1.4. XHTML

XHTML son las siglas de EXtensible HyperText Markup Language. Este lenguaje es una combinación de HTML y XML. XHTML consiste de todos los elementos (en una versión más clara y estricta) de HTML 4.01 combinado con la sintaxis de XML. Las páginas XHTML pueden leerse por todos los dispositivos permitidos para XML, y trabajan en todos los navegadores e incluso con navegadores anteriores. XHTML es un estándar Web ya que XHTML 1.0 se convirtió en una recomendación oficial del W3C el 26 de junio del 2000. XHTML surgió de la necesidad “forzar” que los documentos HTML estén bien formados.

Una DTD XHTML describe con precisión, la sintaxis permitida y la gramática de marcado XHTML.

B.2. Tecnologías relacionadas con XML

Un documento XML no tiene funcionalidad propia, es decir, el documento XML no es un programa que se ejecuta las veces que uno desea. Por ello, además de crear un documento XML se debe escribir un fragmento de software para enviarlo, recibirlo o mostrarlo. Por consiguiente se necesitan herramientas que permitan manipular el lenguaje. Para satisfacer esta necesidad y en respuesta a la aceptación que ha tenido XML en las intranets, han surgido múltiples herramientas para editar, validar y analizar datos XML. En seguida se describen algunas.

B.2.1. DTD

El objetivo de una DTD (Document Type Definition) es definir los componentes básicos legales de un documento (el documento puede ser, por ejemplo, de los siguientes tipos: SGML, XML, XHTML). Define la estructura de un documento con una lista de elementos legales. Una DTD

puede declararse dentro de su documento XML o como una referencia externa. Con las DTD, cada uno de los archivos XML puede contener una descripción de su propio formato consigo.

Las DTD permiten que grupos independientes de personas se pongan de acuerdo en usar una DTD común para intercambiar datos. Una aplicación puede usar una DTD estándar para verificar que los datos que se reciben del mundo exterior son válidos. También se puede usar un DTD para verificar los datos propios.

B.2.2. Esquemas XML

Los esquemas XML son una alternativa a las DTD. Un esquema XML describe la estructura de un documento XML. Al lenguaje de esquemas XML también se le conoce como Definición de Esquema XML (XSD). Fueron propuestos originalmente por Microsoft, se convirtieron en una recomendación oficial del W3C en mayo de 2001.

El objetivo de un esquema XML es definir los componentes básicos legales de un documento XML (como las DTDs). Un esquema XML permite definir características como:

- Los elementos que pueden aparecer en un documento
- Los atributos que pueden aparecer en un documento
- Cuáles elementos son elementos hijo
- El orden de los elementos hijo
- El número de elementos hijo
- Si un elemento es vacío o puede incluir texto
- Los tipos de datos para elementos y atributos
- Los valores por omisión para elementos y atributos

Algunas de las ventajas que los esquemas XML tienen sobre las DTDs son: los esquemas XML son extensibles a futuras adiciones (porque están escritos en XML), dan soporte a tipos de datos y a espacios de nombres. La mayor ventaja de los esquemas XML es el soporte a tipos de datos.

Cuando se envían datos de un remitente a un receptor es esencial que ambas partes tengan las mismas “expectativas” sobre el contenido. Con esquemas XML, el remitente puede describir los datos de forma que el receptor entienda.

B.2.3. CSS

Las etiquetas de HTML al principio fueron diseñadas para definir el contenido de un documento. Como los dos navegadores principales, Netscape e Internet Explorer, siguieron añadiendo nuevas etiquetas de HTML y atributos (p.ej. la etiqueta `` y el atributo `color`) a la especificación de HTML original, se hizo cada vez más difícil crear sitios Web donde el contenido de documentos de HTML fuera claramente separado de la disposición de presentación del documento.

Para solucionar este problema, W3C creó los estilos en HTML 4.0. Tanto Netscape 4.0 como Internet Explorer 4.0 soportan CSS. Los estilos definen como mostrar los elementos de HTML (como la etiqueta de `font` y el atributo `color` en HTML 3.2). Normalmente se guardan en archivos externos a sus documentos de HTML. Los estilos se almacenan en las llamadas *hojas de estilo*. Las *hojas de estilo* externas se almacenan en archivos CSS (Cascading Style Sheets), que

se traduce como Hojas de Estilo en Cascada, esto significa que múltiples definiciones de estilo pueden caer en un solo documento.

Las hojas de estilo externas permiten cambiar el aspecto y la disposición de todas las páginas en un sitio Web solamente editando un único documento CSS.

Cuando un navegador lee una hoja de estilo, formateará el documento de acuerdo a ésta.

B.2.4. XSL

XSL son las siglas de Extensible Stylesheet Language. El XSL es un desarrollo del W3C para cubrir la necesidad de un lenguaje XML basado en hojas de estilo.

Con XSL se puede transformar XML en XHTML, filtrar y ordenar datos XML, definir las partes de un documento XML, formatear datos XML, y escribir datos XML a medios de comunicación diferentes, como pantallas, papel, o voz.

B.2.5. DOM

El Modelo de Objeto para Documento XML (DOM) es una interfaz de programación para documentos XML que define cómo se puede tener acceso a la estructura y el contenido de un documento XML para manipularlo. Como una especificación W3C, el objetivo de DOM XML es proporcionar una interfaz de programación estándar para una amplia variedad de aplicaciones. El DOM XML está diseñado para usarse con cualquier lenguaje de programación y sistema operativo.

Con DOM XML, un programador que creó un documento XML puede recorrer su estructura para añadir, modificar o suprimir sus elementos.

Dado que el DOM oficial no incluye funciones estándar para cargar documentos XML se usa un programa llamado XML parser (analizador gramatical XML) para cargar un documento XML en la memoria de la computadora. Una vez cargado, se puede recuperar su información y manipularla mediante el DOM.

El analizador gramatical de Microsoft (Microsoft XML parser) contiene todas las funciones necesarias para atravesar el árbol de nodos, tener acceso a los nodos y sus valores de atributo, insertar y suprimir nodos, y convertir el árbol de nodos de regreso a XML.

El DOM representa el documento XML en forma de árbol. El nodo `documentElement` es el nivel más alto del árbol. Este elemento tiene uno o muchos `childNodes` que representan las ramas del árbol.

Los elementos XML pueden extraerse de un documento XML recorriendo el árbol de nodos, y accediendo a los elementos por número o por nombre, siendo esta última la que se usa con mayor frecuencia. Una forma común de extraer elementos XML de un documento XML es atravesar el árbol de nodos y extraer el valor de texto de cada elemento.

Una de las grandes promesas de XML es la posibilidad de separar documentos de HTML de sus datos. Usando un analizador gramatical XML dentro del navegador, puede construirse una página de HTML como un documento estático, con JavaScript incrustado para proporcionar datos dinámicos.

B.2.6. SOAP

Actualmente, muchas aplicaciones se comunican usando Llamadas de Procedimientos Remotos (RPC) entre objetos como DCOM Y CORBA, sin embargo, los cortafuegos y servidores proxy normalmente bloquean esta clase de tráfico. HTTP proporciona una mejor opción para comunicar aplicaciones debido a que todos los navegadores y servidores tienen soporte para HTTP. Dado

que HTTP no se diseñó para este propósito, surge SOAP como complemento de HTTP. SOAP permite conectar aplicaciones escritorio de interfaz de usuario gráficas a servidores poderosos de Internet que usan los estándares de Internet HTTP y XML.

SOAP (Simple Object Access Protocol) es un protocolo simple basado en XML que permite a las aplicaciones intercambiar información vía Internet (sobre HTTP). Este protocolo se usa para tener acceso a Servicios Web debido a que es independiente de la plataforma y del lenguaje. Además, al ser basado en XML, es simple y extensible. SOAP se convirtió en una recomendación oficial del W3C en junio de 2003.

Un mensaje SOAP es un documento XML ordinario que contiene los siguientes elementos:

- Un elemento **envelope** (obligatorio) que identifica el documento XML como un mensaje SOAP
- Un elemento **header** (opcional) que contiene información del encabezado
- Un elemento **body** (obligatorio) que contiene información de la llamada y la respuesta
- Un elemento **fault**(opcional) que provee información acerca de los errores que ocurren mientras se procesa el mensaje.

En la figura B.1 se pueden ver gráficamente los elementos de un mensaje SOAP.

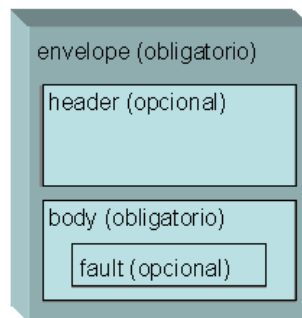


Figura B.1: Partes de un mensaje SOAP

B.2.7. WSDL

WSDL (Web Services Description Language) es un lenguaje que sirve para describir servicios Web. WSDL 1.1 se emitió como una nota W3C² por Ariba, IBM y Microsoft en marzo del 2001. La primera versión no oficial de WSDL 1.2 fue liberada por el W3C en Julio del 2002. Sin embargo WSDL todavía no es un estándar W3C.

Un documento WSDL es un documento escrito en XML que especifica la ubicación de un servicio y las operaciones que ofrece dicho servicio, esto lo hace mediante un conjunto de definiciones. A continuación se listan los principales elementos que se deben incluir en un documento WSDL.

- El elemento **types** define los tipos de datos que usa el servicio Web. Para mantener independencia de la plataforma, WSDL usa la sintaxis de esquemas.

²Una nota W3C se hace disponible por el W3C únicamente para discusión. La publicación de una nota por el W3C no indica la aprobación por el W3C o el grupo W3C, o alguno de sus miembros

- El elemento **message** define los mensajes usados por el servicio Web, define los elementos dato de una operación. Cada mensaje puede tener una o mas partes. Las partes se pueden comparar con los parámetros de una función en un lenguaje de programación tradicional.
- El elemento **portType** define un servicio Web, las operaciones que puede desempeñar dicho servicio y los mensajes que se requieren.
- El elemento **binding** define el formato de los mensajes y detalles del protocolo para cada puerto.
- Un documento WSDL también puede contener otros elementos, como elementos **extension** y un elemento **service** que permite que se agrupen las definiciones de varios servicios Web en un solo documento.

En el apéndice ??WSDL se puede ver una descripción más completa de WSDL y los documentos WSDL de los servicios creados para este proyecto.