Trabajo de Grado
para la carrera de Licenciatura en Informática
de la Universidad Nacional de La Plata

Integración de mecanismos activos en un MBDOO

Autores
Guillermo Avendaño - Fabián Eleno

Directores
Silvia Gordillo - Ana Monteiro

Septiembre de 1995

ÍNDICE

INTRODUCCIÓN

La tecnología de bases de datos ha ido evolucionando desde los sistemas de archivos convencionales hasta el modelo relacional, pasando por el modelo jerárquico y en red. La naturaleza de las aplicaciones de bases de datos ha experimentado cambios rápidos, desde aplicaciones simples hacia aplicaciones complejas. Actualmente, los sistemas de bases de datos relacionales no pueden soportar, ni los requerimientos de productividad en el desarrollo de aplicaciones complejas, ni los requerimientos de performance al ejecutar tales aplicaciones [Kim95].

En los últimos años han tomado cuerpo las investigaciones sobre Bases de Datos Orientadas a Objetos (BDOO). Las BDOO han sido introducidas en el mercado, parcialmente en respuesta al crecimiento anticipado en el uso de lenguajes de programación orientados a objetos, y como un intento de mejorar algunas de las deficiencias de los sistemas de bases de datos relacionales, que surgen de las restricciones inherentes al modelo de datos relacional.

Los sistemas de bases de datos son herramientas que los diseñadores usan en el desarrollo de aplicaciones, para resolver algún problema específico. Actualmente los sistemas de bases de datos, sobre todo los relacionales y aún orientados a objetos, carecen de alguna de las principales características necesarias para el desarrollo en algunos dominios de aplicaciones. Por ejemplo, facilidades para manejo de datos multimediales, manejo de transacciones de larga duración, manejo de datos espaciales, control automático de restricciones, etc. [Kim95]. Si un sistema de bases de datos carece de alguna capacidad importante, los diseñadores de la aplicación deben implementarla dentro de la misma.

Toda base de datos es un modelo de algún sistema del mundo real. El contenido de la misma es visto como una foto de un estado en el entorno de la aplicación, y cada cambio en la base de datos debería reflejarse como un evento en ese entorno [McCar89].

Cuando una base de datos se diseña conceptualmente, uno de los principales esfuerzos está destinado a capturar todas aquellas interacciones y restricciones que deben cumplir los datos [CeWi92]. Desafortunadamente, debido a la carencia de una tecnología apropiada, una menor parte de este esfuerzo se vuelca en la definición de la base de datos. En consecuencia, dichas restricciones deben ser tenidas en cuenta, y controladas desde las aplicaciones.

Las interacciones y restricciones que deben mantener los datos para que el estado de la base sea válido pueden expresarse en algún formalismo, como por ejemplo el lógico (lógica proposicional, de primer orden, de Horn, etc). Luego, es necesario que el DBMS pueda interpretarlas, y extender su funcionalidad con características activas y/o deductivas, no provistas por los DBMS convencionales.

Los sistemas de bases de datos convencionales son pasivos: sólo ejecutan consultas o transacciones explícitamente solicitadas por un usuario o un programa de aplicación. Para varias aplicaciones, sin embargo, es importante monitorear situaciones de interés obteniendo una respuesta oportuna cuando ellas ocurren [Kim95]. Por ejemplo, para controlar que las restricciones definidas sobre los datos no sean violadas, deben incluirse en el código de las aplicaciones aquellas operaciones necesarias para evaluar y restaurar la consistencia de la base. Esto implica una sobrecarga de requerimientos en los programas de aplicación, ya que tendrán que proveer el código necesario, adicional al propio, para chequear la consistencia de los datos.

Un DBMS activo responde a eventos generados independientemente de la solicitud de una aplicación, ejecutando acciones automáticamente cuando se alcanzan ciertas condiciones. El sistema monitorea la ocurrencia de ciertos eventos y cuando alguno se produce, ocasiona el chequeo de un conjunto de expresiones que representan el estado correcto de los datos, o configuraciones de estados de la base.

En el primer caso, cuando se alcanza un estado inválido se ejecuta alguna operación de restauración. En el segundo, estas configuraciones describen condiciones en las cuales debe ejecutarse alguna tarea específica.

Las bases de datos activas incorporan capacidades que permiten realizar control automático de restricciones, conocido como *Imposición automática de Consistencia.* Las características activas pueden utilizarse también para restringir, por ejemplo, el estado de un objeto, el estado de un objeto con respecto a otros, el acceso desde un usuario o una aplicación a determinados objetos, la versión del objeto que está vigente en un momento dado, etc. De todas maneras, cada una de ellas está describiendo un problema particular que es materia de investigación. De allí surge el uso para gerenciamiento automático de versiones, políticas de acceso y seguridad, y otros [EiWe93], [CeWi92].

Las facilidades activas pueden utilizarse tanto para implementar funciones propias de un DBMS, como para extenderlas. Las restricciones de integridad referencial y de composición son mecanismos provistos por la mayoría de los OODBMS, y que pueden ser llevados a cabo con la incorporación de mecanismos activos. También son útiles para efectuar el mantenimiento de valores derivados, una de las características de modelización que incorporan las bases de datos orientadas a objetos. Por otro lado, pueden usarse para extender las capacidades convencionales, por ser adecuadas para realizar un control automático de restricciones de integridad semántica, ejecutar operaciones automáticas, en función del estado de la base, sin la necesidad de una invocación explícita, etc.

Una base de datos activa requiere un mecanismo de control sobre los cambios de estados de la base, y un motor de ejecución mucho más poderoso de los que ofrecen los DBMS convencionales. Según Stonebraker[Ston92], la incorporación de reglas y mecanismos de activación de reglas parece ser la propuesta obligada cuando se intenta construir un DBMS activo. Los sistemas de reglas son una herramienta adecuada para disparar acciones automáticamente (correctivas o pasos en la solución de un problema), cuando ciertas condiciones se vuelven verdaderas.

Lógicamente, la incorporación de un sistema de este tipo torna la semántica de ejecución más complicada [WiFi90], ya que varias cuestiones tienen que ser resueltas, por ejemplo: en qué momento se evalúan las reglas, cuáles reglas deben evaluarse, qué ocurre si más de una regla puede ser ejecutada, etc.

Es necesario, en este contexto, estudiar técnicas de optimización tanto para evaluar las condiciones, como para decidir qué regla ejecutar desde un conjunto de reglas posibles. Por otro lado, debe definirse cómo administrar los puntos de interacción entre el flujo normal de un programa y la activación de reglas.

El objetivo del presente trabajo es mostrar los aspectos relevantes para extender la definición de SIGMA[1] a un modelo de base de datos activo.

En el capítulo uno se revisan los conceptos básicos sobre sistemas de bases de datos orientados a objetos, en el capítulo dos se describen las ventajas de contar con un DBMS que tenga capacidad para realizar el control automático de restricciones, justificando la necesidad de la incorporación de un sistema de reglas para tornarlo en un modelo activo. En los capítulos tres y cuatro se exponen los sistemas de reglas, sus principales características y funcionamiento, y las cuestiones que deben resolverse cuando se los incorpora en un DBMS. En el capítulo cinco se definen los aspectos fundamentales para incorporar un sistema de reglas en SIGMA. En el seis se hace un análisis de las propuestas existentes más difundidas sobre bases de datos activas, comparando sus principales características con las del capítulo cinco. Por último, en el séptimo, se realizan las conclusiones del trabajo.

---

[1] Modelo unificado para Bases de Datos Orientadas a Objetos definido en un Trabajo de Grado anterior [SmTa93].

CAPÍTULO 1
CONCEPTOS BÁSICOS DE BDOO

I. Características de los sistemas de bases de datos convencionales

Los sistemas de bases de datos proveen distintos lenguajes para permitir a los programadores de una aplicación, o usuarios finales, definir y manipular una base de datos. Un lenguaje de base de datos convencional consiste de tres componentes:

a) Un *lenguaje de definición de datos*(DDL): permite la especificación del esquema de la base de datos, que comprende la definición de las entidades con sus interrelaciones y restricciones.

b) Un *lenguaje de manipulación de datos*(DML): provee las facilidades para expresar consultas (queries), y actualizaciones sobre la base de datos.

c) Un *lenguaje de control de datos*(DCL): incluye facilidades para proteger la integridad de la base y para administrar los recursos del sistema.

Un DCL incluye transacciones como una forma limitada de restricciones de integridad semántica y autorización. A menudo varias operaciones de bases de datos forman una unidad de trabajo simple. Una transacción es una colección de operaciones que realiza una función lógica en una aplicación de bases de datos. Cada transacción es tratada como una operación atómica. Si una transacción no termina exitosamente, el sistema automáticamente deshace cualquier cambio realizado en la base por la transacción. En cambio, si termina exitosamente, todas las escrituras dentro de la base de datos se vuelven permanentes. Las restricciones de integridad semántica son condiciones impuestas sobre el contenido de la base. A causa del costo de rendimiento (perfomance) en las actualizaciones, casi ningún sistema soporta chequeo automático de restricciones de integridad.

Además de las características expresadas en los lenguajes de base de datos, los sistemas proveen facilidades para control de concurrencia, recuperación de fallas, etc. El objetivo del control de concurrencia es proteger la integridad de la base cuando varios usuarios acceden simultáneamente al mismo dato. La recuperación de fallas es necesaria para restaurar la base a un estado consistente luego de caídas del sistema, ya sean estas producidas por errores de soft o de hard.

II. Características adicionales para soportar los nuevos requerimientos

La complejidad de las aplicaciones actuales requiere que los sistemas de bases de datos se adapten y agreguen nuevas características para soportarlas. La lista de tales características es extensa, y dependerá de cual sea la aplicación. Se pueden mencionar las siguientes:

a) capacidad para representar objetos anidados, para permitir el refinamiento sucesivo de entidades complejas,

b) capacidad para recuperar y manipular datos arbitrariamente complejos como los que necesitan las aplicaciones multimediales,

c) capacidad para manipular tipos de datos arbitrarios,

d) administración de cambios en el tiempo de la base de datos (versiones de: un objeto, del esquema, etc.),

e) capacidad para especificar reglas y restricciones extendidas para soportar manejo de inferencia, necesarias en las aplicaciones basadas en conocimiento,

f) capacidad para manejar transacciones cooperativas de larga duración, etc.

III. Bases de datos orientadas a objetos

Una base de datos orientada a objetos (BDOO) es una colección de objetos cuyo comportamiento, estado, y relaciones se definen respetando un modelo orientado a objetos [Kim91]. Los conceptos orientados a objetos forman una base apropiada para satisfacer los requerimientos de las nuevas aplicaciones de bases de datos. Además, estos manejadores de bases de datos incluyen todos los conceptos de los sistemas convencionales (lenguajes de consulta, persistencia, distribución, etc.). Un modelo orientado a objetos permite representar no solo datos complejos y relaciones entre ellos,

sino que provee además encapsulamiento de datos y programas, soporta la creación de nuevos tipos de datos, y elimina el problema de incompatibilidad(impedance mistmatch) existente en los modelos relacionales.

Los conceptos orientados a objetos que han sido implementados en distintos lenguajes de programación no adoptan la misma interpretación, por lo tanto no existe un consenso universal acerca del modelo de datos orientado a objetos. Sin embargo, pueden extraerse un conjunto de nociones generales que caracterizan al paradigma orientado a objetos.

III.1 Conceptos del modelo orientado a objetos

Para definir conceptos orientados a objetos se necesita comprender primero que es un objeto: un *objeto* es una entidad que encapsula propiedades y comportamiento. Es una máquina abstracta que define un protocolo a través del cual es posible interactuar. El protocolo se establece por medio de mensajes implementados por métodos. Un objeto es instancia de un tipo de objetos.

Un *tipo de objetos* es un molde donde se especifican el comportamiento y las propiedades(o atributos) de los objetos que modeliza. El comportamiento es implementado por un conjunto de métodos, con los cuales los objetos responden a mensajes.

Las instancias correspondientes a un tipo se agrupan en colecciones de objetos llamadas *clases*.

El mecanismo de *herencia* provee una modificación incremental de las definiciones de los tipos, permitiendo que los nuevos tipos compartan las definiciones existentes. La orientación a objetos se complementa con los conceptos de encapsulamiento, herencia y polimorfismo. *Polimorfismo* es el mecanismo por el cual objetos de distinto tipo pueden responder de diferente manera al mismo mensaje.

Cuando se define un tipo, se lo hace como subtipo de uno o más tipos existentes estableciendo una relación *es_un* entre ellos. Así se conforma una estructura jerárquica de tipos representado como un grafo acíclico dirigido (D.A.G.). La *jerarquía de tipos* determina también una jerarquía de clases, que establece una relación de inclusión entre los conjuntos de objetos de una clase, y los de una superclase. De este modo, la raíz de

la jerarquía de tipos se corresponde con la clase que contiene a todos los objetos de la base de datos.

Otro concepto importante que soportan los sistemas orientados a objetos es el de *composición de objetos,* este mecanismo de modelización establece una relación denominada *parte_de*, en la cual los objetos referenciados son llamados *componentes*.

Los conceptos de orientación a objetos están fundamentalmente diseñados para reducir las dificultades de desarrollo y evolución de sistemas de software complejos. El encapsulamiento y herencia permiten que los tipos puedan ser reusados como base para construir bases de datos y programas complejos y, por otro lado, el uso del polimorfismo garantiza la genericidad del código. Este es precisamente el objetivo que ha buscado la tecnología de manejo de datos, desde los sistemas de archivos hasta los sistemas de bases de datos relacionales en las últimas tres décadas.

En resumen, el paradigma de orientación a objetos reduce el *gap* semántico que existe entre la concepción del problema del mundo real y la representación computacional del mismo.

A pesar de todos los beneficios que incorporan las bases de datos orientadas a objetos en el desarrollo de software, la mayoría de los OODBMS no cuentan aún con un mecanismo para realizar el control automático de las restricciones semánticas que deben cumplir los objetos integrado al modelo mismo, aunque el problema de control automático de restricciones, llamado imposición automática de consistencia, es anterior al surgimiento de las bases de datos orientadas a objetos.

CAPÍTULO 2
RESTRICCIONES E IMPOSICIÓN AUTOMÁTICA DE CONSISTENCIA

El desarrollo de un sistema de información, soportado por una base de datos, puede verse como el proceso de construir un modelo de un problema del mundo real.

En la definición del esquema de una base de datos (BD) deben conceptualizarse las entidades que intervienen en las aplicaciones y, además, capturarse todas las interacciones y restricciones que deben cumplir las mismas para que la configuración de la BD tenga sentido. Estos requisitos, que siempre deben cumplir las entidades, se pueden expresar por medio de predicados lógicos, que representan un estado consistente de la BD.

Desafortunadamente, debido a la carencia de una tecnología apropiada, en el diseño de la base de datos sólo es posible capturar restricciones estáticas muy simples como unicidad de claves o integridad referencial [CeWi92].

Es común que sobre una misma base de datos existan varias aplicaciones que accedan, desde sus programas, para consultarla o actualizarla. Un programa que modifique una o más entidades de la base deberá controlar que las restricciones definidas sobre ella no dejen de valer. Este control implica el chequeo de las condiciones que representan aquellas restricciones, y si se descubre un estado inconsistente, entonces deben tomarse acciones que restauren la consistencia, ya sea cancelando la transacción corriente, o ejecutando operaciones correctivas.

El control de las restricciones sobre la base es una tarea que se repite para cada programa que deba actualizarla. Así, el mantenimiento de la consistencia de la BD se vuelve más una cuestión de disciplina de programación, que una propiedad inherente al esquema de la base de datos [CFPT93].

Un modo de resolver el problema sería mediante un módulo general que haga una abstracción del chequeo de las restricciones, para que sea usado por todos los programas que corran sobre la base de datos. De esta manera se libera al programador de conocer como evaluar las condiciones definidas sobre la base, y de como restaurar un estado consistente cuando se ha producido uno inconsistente. Sin embargo, esta solución resulta limitada: los programas que utilizan ese módulo no quedan exentos de hacer el control explícitamente. Como consecuencia, si se agregan o eliminan restricciones, los programas igualmente deben ser modificados.

Una solución propuesta por varios autores [Han92] [McCar89] [ScCa92], es proveer al DBMS de mecanismos que permitan controlar automáticamente las restricciones, desligando a los programas de tales chequeos. Interesantes ventajas son obtenidas con esta propuesta:

1 - Se acentúa la modularidad en el desarrollo de las aplicaciones, ya que se elimina el chequeo de las restricciones del código de los programas.

2 - Como las restricciones son inherentes a las entidades más que a las aplicaciones, resulta más natural que sean definidas y controladas junto con las entidades, por el DBMS.

I - Imposición automática de consistencia

Las restricciones que se especifican sobre las entidades de una base de datos son llamadas *restricciones de integridad semántica* [EiWe93]. Anteriormente se vio que estas juegan un rol importante en la tarea de diseño y desarrollo de software. Por otro lado, una operación que viole cualquiera de esas restricciones debe ser evitada o corregida. Para corregir cualquier posible estado inconsistente de la base, el diseñador debe definir las operaciones que la restauren a un estado consistente. Tales operaciones comúnmente son llamadas *manejadores de excepción*.

Es deseable que el DBMS provea métodos para especificar las restricciones de integridad y un mecanismo para controlarlas automáticamente. Ese mecanismo debería

ser capaz de desviar el flujo de control de los programas a los manejadores de excepción que chequean las restricciones, y ejecutar las acciones de corrección necesarias, si un estado inconsistente fuera alcanzado.

No contar con un mecanismo que permita controlar las restricciones, y realizar la imposición "automática" de consistencia, origina una serie de inconvenientes [EiWe93] :

1 - Programas de aplicación con gran cantidad de líneas de código: además del código que corresponde a la operación los programadores deben proveer código para el chequeo de restricciones.

2 - Gran cantidad de código redundante: diferentes programas de aplicación tienen que chequear restricciones iguales.

3 - Dificultad ante el cambio de restricciones: si una restricción es cambiada, varios programas deben ser reescritos.

4 - Poca confiabilidad: usualmente las bases de datos coleccionan entidades que son usadas por varias aplicaciones. Además, distintas aplicaciones generalmente son desarrolladas por diferentes equipos de trabajo. Si la imposición de consistencia debe ser realizada en los programas de aplicación, puede suceder que el código que chequea una restricción dentro de un programa sea omitido, intencionalmente o no. En tal caso el DBMS no detecta ni evita tal situación, provocando que el error sea trasladado a otras aplicaciones que usan la base.

5 - Ineficiencia en el desarrollo de los programas: los programadores tienen que concentrarse, no sólo en escribir los requerimientos de la transacción que están programando, sino que además deben dispersarse de su objetivo para tener que escribir, y pensar, en el chequeo de las restricciones.

Para que un DBMS realice la imposición automática de consistencia debe soportar algún mecanismo que esté 'atento' para detectar que una actualización, realizada sobre la base no viole ninguna restricción. Tales mecanismos pueden encuadrarse en el paradigma de programación basada en reglas, donde una regla

consiste de una parte de condición (antecedente) y otra de acción o conclusión (consecuente).[Dur94]

II - Mecanismo de reglas

Para Stonebraker[Ston92], un sistema de reglas que interactúe con el sistema de transacciones del DBMS parece ser una propuesta, casi obligada, para alcanzar la imposición automática de consistencia.

Una regla consiste de una condición, que expresa un estado de la base de datos, y una acción (operación sobre la base). La semántica de ejecución de un DBMS, con un soporte para reglas, es sencilla: cuando la condición de una regla se satisface, significa que la BD ha caído en un estado inconsistente y, en consecuencia se dispara la acción correspondiente para 'restaurar' la consistencia de los datos.

En el campo de la Inteligencia Artificial se han estudiado sistemas de reglas para construir sistemas expertos. Un sistema de reglas consiste en: una *memoria de trabajo*, que captura los hechos conocidos del problema, una *base de conocimiento* o memoria de reglas, donde se mantiene el conjunto de reglas, y una *máquina de inferencia*, que combina hechos del problema contenidos en la memoria de trabajo, con las reglas de la base de conocimiento, para inferir nueva información.

El modelo de ejecución de un sistema basado en reglas consiste en introducir cambios en la memoria de trabajo, ocasionando que se instancie alguna regla, cuya acción producirá nuevos cambios, provocando la posible activación de otras reglas. Una característica destacable de estos sistemas es que su comportamiento depende exclusivamente del estado de la memoria de trabajo.

Un DBMS, con un sistema de reglas incorporado en su arquitectura, tiene la capacidad de responder automáticamente con operaciones no solicitadas explícitamente por un usuario, o una aplicación. Tales sistemas son conocidos como *DBMS activos*. Las capacidades activas de estos sistemas posibilitan la imposición automática de consistencia, el manejo de datos derivados, el control de autorizaciones, etc.

Las bases de datos activas deben extender el modelo de ejecución de los DBMS convencionales, basado en transacciones, con el de los intérpretes de los sistemas de reglas. La extensión puede hacerse integrando ambos modelos de ejecución en el DBMS o mediante un módulo externo al mismo. [McCar89] [CET93]

CAPÍTULO 3
INCORPORANDO UN SISTEMA DE REGLAS EN UN DBMS

En el capítulo anterior se presentó un área de interés común para los investigadores sobre bases de datos e inteligencia artificial: la idea de incorporar un mecanismo de reglas en un DBMS, justificados en el beneficio de proveer medios para asegurar, por ejemplo, la imposición automática de consistencia.

Cabe destacar que la diversidad de propuestas existentes sobre sistemas de reglas ofrecen, también, distintas características cuando se los incorpora en un DBMS. Dicha integración da origen a nuevos paradigmas de base de datos, que varían de acuerdo al tipo de sistema de reglas usado.

En este capítulo se describen los puntos relevantes de los sistemas de reglas en general, y se muestran dos paradigmas de bases de datos que surgen al incorporarlos en un DBMS: las bases de datos activas y las bases de datos deductivas.

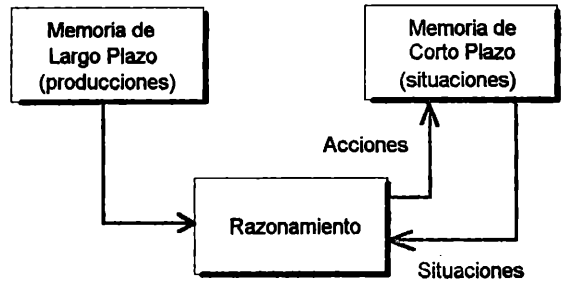
I - Origen de los sistemas de reglas

La idea de los sistemas de reglas proviene de los *sistemas de producción*.[Dur94] Producción es el término usado en psicología cognitiva para describir relaciones entre situaciones y acciones. En la década del '60 Newell y Simon estudiaron la aplicación de técnicas de procesamiento de información para modelizar la forma en que los seres humanos resuelven problemas. En su trabajo representaron la memoria de larga duración del hombre como un conjunto de reglas situación/acción, llamadas

producciones, y la memoria de corta duración como un conjunto de situaciones, o información específica acerca de un problema.

Ellos argumentaron que cuando el hombre resuelve algún problema usa un conjunto de producciones desde su memoria de largo plazo, que aplica a alguna situación almacenada en su *memoria de corto plazo.* La situación causa que se dispare alguna producción, provocando que nuevas acciones sean agregadas a la memoria de corto plazo.

Este proceso describe una forma de razonamiento humano: inferir nueva información desde información conocida. Con esta información adicional agregada a la memoria de corto plazo la situación cambia, pudiendo causar que se disparen otras producciones. Este modelo de evocar producciones desde la memoria de largo plazo, y cambiar el contenido de la memoria de corto plazo, se conoce como *sistema de producción* [Dur94].



Modelo de los Sistemas de Producción

Los sistemas de producción son el fundamento de los actuales sistemas de reglas.

II - Sistemas de reglas

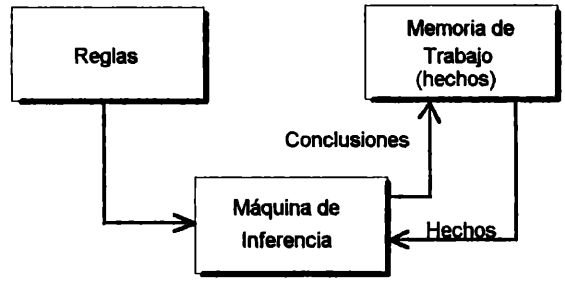
Una regla es una estructura de conocimiento que relaciona alguna información conocida, a otra que puede ser concluida o inferida. Ellas asocian determinada información, con alguna acción. Esta acción puede ser la conclusión de alguna nueva información o la ejecución de algún procedimiento [Dur94].

Una regla consiste de un 'antecedente', comúnmente llamado premisa, contenido en la parte **IF,** o parte izquierda (LHS) y un 'consecuente', también llamado conclusión, contenido en la parte **THEN,** o parte derecha (RHS).

La premisa de una regla describe una configuración de la memoria de trabajo, que cuando se satisface, indica que debe ejecutarse, o que la conclusión es válida.

En un sistema basado en reglas de producción, el procesamiento de las reglas es administrado por un módulo conocido como la *máquina de inferencia*.

La máquina de inferencia actúa como el módulo de razonamiento del modelo de sistemas de producción, comparando los hechos con los antecedentes de las reglas, para ver cuales pueden dispararse. Las reglas disparadas introducen nueva información en la memoria de trabajo, haciendo que otras puedan ser activadas. El proceso continúa hasta que no haya reglas que se satisfagan con la información de la memoria de trabajo.



Modelo de los Sistemas de Reglas

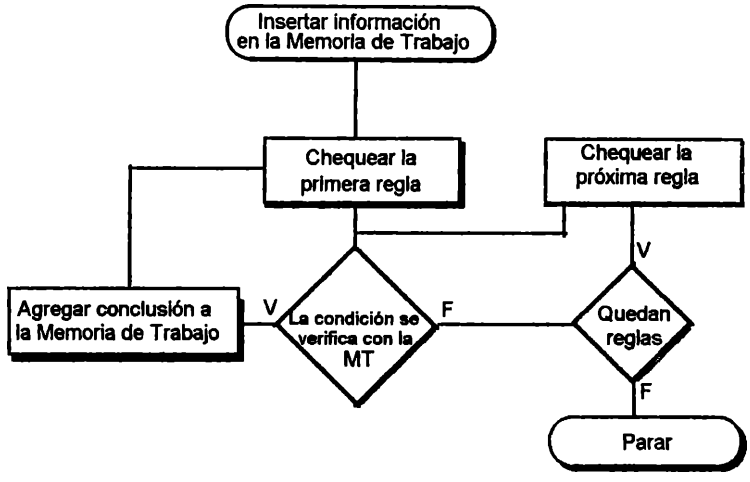
Los sistemas de reglas procesan la información contenida en su memoria de trabajo, con un conjunto de reglas contenido en su base de conocimiento, usando una máquina de inferencia para derivar nueva información.

Las reglas pueden ser aplicadas hacia adelante o hacia atrás según el tipo de estrategia que utilice la máquina de inferencia. A partir de esto, se originan dos tipos de sistemas de reglas.

II.1 - Reglas con encadenamiento hacia Adelante (Foreward Chaining)

En los sistemas de reglas donde la máquina de inferencia usa una estrategia de encadenamiento hacia adelante, se parte de un conjunto de hechos conocidos, contenidos en la memoria de trabajo, luego se derivan nuevos hechos usando reglas cuyas premisas coincidan con los hechos conocidos("matching"), y se continúa este

proceso hasta que un 'objetivo' es alcanzado o no queden reglas cuyas premisas coincidan con hechos conocidos o derivados.



Proceso de Inferencia hacia adelante

El sistema evalúa secuencialmente los antecedentes de las reglas, hasta encontrar una que se satisface. Al encontrarla, ejecuta su parte derecha. Sin embargo, puede haber otras reglas que se verifiquen con el actual contenido de la memoria de trabajo, y el sistema no las tuvo en cuenta. Además, la regla ejecutada puede no haber sido la más conveniente. Por lo tanto, es necesario contar con alguna estrategia para resolver este conflicto.

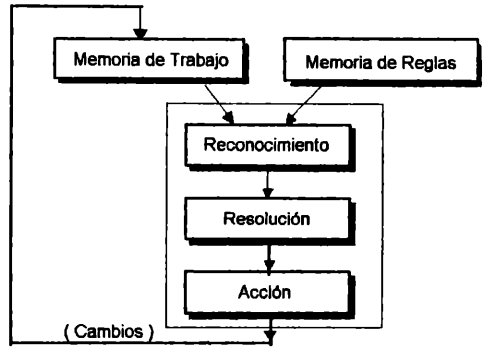
La incorporación de un módulo para resolver cual regla disparar, cuando varias compiten, da origen a un nuevo paso en el proceso de inferencia anterior, llamado *Resolución de Conflicto.*

Con la incorporación de la resolución de conflicto, la máquina de inferencia queda como un proceso cíclico de tres pasos: Reconocimiento-Resolución-Acción.

1 - Reconocimiento: consiste en la comparación de los antecedentes de las reglas con la información de la memoria de trabajo. Como resultado se obtiene el conjunto de reglas que se pueden disparar (conjunto de conflicto).

2 - Resolución: elección de la regla a disparar desde el conjunto de conflicto, siguiendo alguna estrategia.

3 - Acción: ejecutar la parte derecha de la regla elegida.



Maquina de Inferencia con resolución de Conflicto

La estrategia de inferencia con encadenamiento hacia adelante (Foreward Chaining) es una buena técnica para tratar problemas donde, a partir de una configuración inicial de la información, se llega a conclusiones, las que pueden ser tanto una acción (procedimiento), o una deducción.

Un sistema de reglas que frente a un estímulo, como por ejemplo nueva información agregada en la memoria de trabajo, tiene determinadas reacciones, posee la característica de ser un *sistema activo.*

II.2 - Reglas con encadenamiento hacia Atrás (Backward Chaining)

En un sistema donde la máquina de inferencia usa una estrategia de encadenamiento hacia atrás, se comienza con un objetivo (inicial) a probar. Primero, se chequea la memoria de trabajo para ver si el objetivo esta presente en ella. Si el objetivo no se encuentra, el sistema busca una regla que lo contenga en su conclusión. El antecedente de esa regla pasa a ser el nuevo objetivo a probar. Dicho antecedente puede estar compuesto de varias premisas, las que se vuelven subobjetivos a probar. Este proceso continúa de una manera recursiva, hasta que se encuentra cada uno de los subobjetivos en la memoria de trabajo [Dur94].

Este formalismo de reglas es apropiado para resolver problemas donde se necesita probar teoremas. En el área de bases de datos se usan para conseguir *procesadores de queries* más sofisticados, dando lugar al paradigma de *bases de datos deductivas.*

III - Intérpretes de sistemas de reglas

Los intérpretes de sistemas de reglas implementan la máquina de inferencia. Son algoritmos que buscan disminuir el costo computacional de la exploración, que se necesita para detectar las reglas cuyas condiciones son satisfechas por el estado de la memoria de trabajo.

En general, estos intérpretes representan las condiciones de las reglas como una conjunción o disyunción de *patrones*. Los patrones son predicados simples como: igual(x,y), mayor(x,y), etc.

El conjunto de conflicto se implementa como una colección de pares ordenados de la forma <Regla de producción, Lista de elementos que hacen verdadera la condición>. Cada par ordenado representa una instanciación de una regla.

Dos de los algoritmos más conocidos de estos intérpretes son RETE[For82] y TREAT[Mira90], están explicados en el Anexo 1. En ellos, la información de cambios en la memoria de trabajo que ingresa al intérprete se denomina *Token.*

Para reducir el costo computacional, los algoritmos comparan un conjunto de condiciones de reglas contra un conjunto de elementos presentes en la memoria de trabajo, sin iterar sobre los conjuntos.

IV - Incorporando un sistema de reglas en un DBMS

La estrategia de encadenamiento que utiliza la máquina de inferencia determina la semántica de procesamiento de las reglas. Las investigaciones sobre la incorporación de procesamiento de reglas en sistemas de bases de datos se ha dividido históricamente en dos áreas: bases de datos deductivas y bases de datos activas.

En las bases de datos deductivas reglas del estilo de programación lógica, con encadenamiento hacia atrás y backtraking, se usan para proveer una interface más poderosa que las que brindan casi todos los lenguajes de queries [Wid93].

En las bases de datos activas un estilo de reglas de producción, con encadenamiento hacia adelante y sin backtraking, son usadas para soportar ejecución automática de operaciones sobre base de datos, en respuesta a eventos determinados, cuando se cumplen ciertas condiciones [CeWi92].

Los paradigmas de bases de datos que surgen desde la integración de sistemas de reglas en un DBMS, bases de datos deductivas y bases de datos activas, más que una división de paradigmas, describen distintas funcionalidades de una nueva tecnología de bases de datos [Wid93].

Las capacidades activas están orientadas a resolver problemas de actualización de las entidades, dando por ejemplo una aproximación justa a la imposición automática de consistencia. Por otro lado, las características deductivas son apropiadas para soportar procesadores de consultas (Queries) más sofisticados, haciendo posible sistemas con capacidad deductiva que resuelva, por ejemplo, consultas recursivas. En [Wid93] se plantea que las funcionalidades activas y deductivas de los DBMS no constituyen una división de paradigmas, sino los extremos de un espectro de posibles capacidades a incorporar, con la integración de un sistema de reglas, en el DBMS.

También existen argumentos para decir que ambas funcionalidades no son disjuntas y, por lo tanto, pueden ser soportadas en un mismo DBMS. En [DUHK94] se hace una propuesta de bases de datos activas, deductivas y orientadas a objetos.

Con la introducción de un sistema de reglas en un DBMS el flujo de control de una transacción es administrado por dos modelos de ejecución, cuya interacción debe estar bien definida. La naturaleza no secuencial de la ejecución de los sistemas de reglas debe sincronizarse con la ejecución secuencial del sistema de transacciones del DBMS.

¿Cómo se reconocen las situaciones donde se hace necesario chequear las reglas?, ¿qué acción toma el sistema cuando se encuentra tal situación, cómo funciona el sistema de reglas?, ¿qué estrategia usa para determinar si una condición de regla se ha vuelto verdadera?, etc., son algunas de las cuestiones que se plantean durante la construcción de un DBMS activo.

CAPÍTULO 4
CONSTRUCCIÓN DE UN DBMS ACTIVO

El propósito principal de un DBMS es llevar a cabo transacciones solicitadas desde un programa de alguna aplicación. Una transacción es una unidad de trabajo sobre la base de datos [Date90].

En un DBMS convencional, una transacción sobre la base se realiza siguiendo secuencialmente los pasos especificados en el programa que está ejecutándose. Al terminar la transacción, si es exitosa, los cambios que produjo son volcados en la base, caso contrario es abortada, siendo su efecto nulo.

Casi todos los modelos permiten definir un conjunto de acciones de control de consistencia, el cual es fijo para cada transacción. Esto provee una capacidad de compensación automática cuando la base cae en un estado inconsistente, y saca provecho de la propiedad de atomicidad de la transacción, pero resulta una política muy poco flexible para un control automático de restricciones complejas.[DHL90]

Con la introducción de un sistema de reglas, que torna al DBMS en activo, se exploran modos más flexibles para expresar el flujo de control, incluyendo operaciones de compensación para transacciones que lleven la base a un estado inconsistente.

Agregando reglas para controlar las actividades sobre la base, el sistema puede automáticamente disparar procesamiento adicional cuando un evento, o situación de interés tenga lugar, permitiendo la especificación del flujo de control de una manera más modular, necesario para chequeo de restricciones de integridad, manejo de condiciones de excepción, etc. [DHL90].

En el capítulo anterior se mostró por qué los sistemas de reglas son un paradigma de programación adecuado para incorporar en los sistemas de bases de datos, para que éstos adquieran la capacidad de ser activos.

El punto inicial de estudio, para la construcción de un DBMS activo, es la integración de dos modelos de ejecución: el de los sistemas de reglas, y el de los sistemas de transacciones de los manejadores de bases de datos. En este capítulo se hace un análisis de como deben interactuar dichos modelos, y se muestra el surgimiento de una nueva estructura de reglas, propias de las bases de datos activas, llamadas reglas evento-condición-acción. [McCar89]

Tales reglas resultan un mecanismo poderoso y uniforme para un número de tareas de bases de datos útiles. Ellas pueden imponer restricciones de integridad, implementar "triggers" y "alerters", mantener datos derivados, imponer restricciones de acceso, implementar políticas de control de versión, etc. Además, el poder de inferencia de las reglas de producción hace que los sistemas de bases de datos activas sean una plataforma adecuada para construir sistemas expertos, y bases de conocimiento grandes y eficientes. [Kim95]

I - Sistema de Transacciones

Una transacción es una porción de un programa, correspondiente a alguna aplicación, que realiza una secuencia de lecturas y escrituras en una base de datos. Una transacción tiene cuatro propiedades, comunmente identificadas como ACID:

'atomicity' significa que la secuencia de lecturas y escrituras en una transacción es vista como una acción simple frente a la base de datos.

'consistency' asegura que la base de datos queda en un estado consistente luego de la ejecución de una transacción.

'isolation' requiere que cada transacción 'observe' un estado consistente de la base de datos, por ejemplo, no lea resultados intermedios de otra transacción.

'durability' requiere que los resultados de una transacción que terminó correctamente se hagan permanentes.

Un paso de una transacción que realice una lectura o escritura sobre la base de datos está haciendo un requerimiento al DBMS. En un OODBMS dicho paso es un

mensaje aplicado a un objeto, al cual se lo llama *transacción atómica de alto nivel*. En los sistemas orientados a objetos se utiliza el concepto de transacciones anidadas. En este modelo cada transacción consiste de subtransacciones atómicas, las cuales también pueden ser anidadas, formado un árbol de transacciones. Para que los cambios de una transacción se registren en la base, cada subtransacción de la misma debe haber terminado exitosamente.

Las actualizaciones del árbol de transacciones se vuelven visibles solo después que la transacción de alto nivel se ha registrado. Las actualizaciones de una transacción de nivel intermedio, se vuelven visibles sólo dentro del alcance de su predecesor inmediato [Date90].

II - Integrando un sistema de reglas con el sistema de transacciones

Un intérprete de un sistema de reglas determina si hay alguna regla cuya condición se satisface como consecuencia de las modificaciones realizadas por algún paso de una transacción. Si encuentra una regla que se satisface, solicita al sistema de transacciones que ejecute la acción correspondiente a la condición de esa regla. Cuando la ejecución de la acción termina, el control vuelve al sistema de reglas, repitiéndose el paso anterior hasta que no haya ninguna acción de regla para ejecutar. Luego el control vuelve al sistema de transacciones, que retoma la ejecución en el punto donde la interrumpió al darle el control al sistema de reglas.

II.1 - Modelo de transacciones vs. Intérpretes de sistemas de reglas

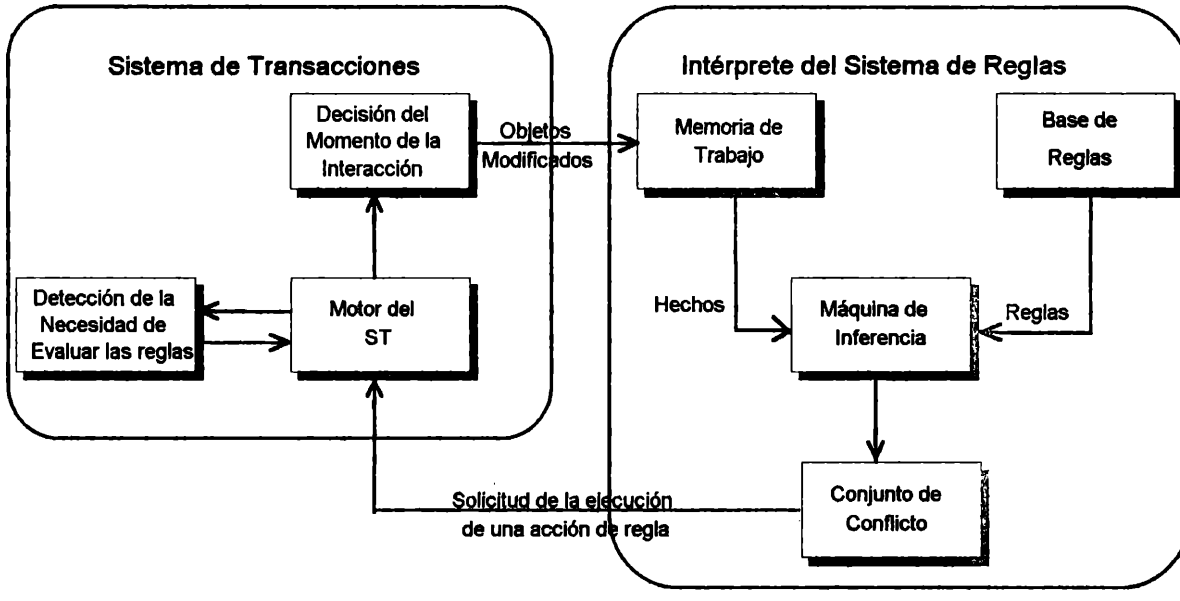
Es claro, entonces, que el modelo de ejecución de un sistema de base de datos activa consta de dos módulos: el sistema de transacciones, que lleva a cabo las operaciones sobre la base de datos, y el intérprete de sistemas de reglas, que determina las acciones que se deben tomar, de acuerdo a las reglas especificadas en la base de datos.

La interacción entre el sistema de transacciones y el intérprete del sistema de reglas (SR) tiene tres aspectos principales, cuya definición caracteriza el funcionamiento del DBMS activo.

Primero, ¿cómo se detecta la necesidad de evaluar las reglas? El sistema de transacciones tiene que estar 'atento' para descubrir que la ejecución de una transacción ha pasado por un lugar que 'señala' la necesidad de pasarle el control al intérprete del SR.

Segundo, habiéndose detectado que deben chequearse las reglas, ¿en qué momento se pasa el control al intérprete?

Y tercero, ¿qué información recibe el intérprete para representar un cambio en la memoria de trabajo?



La interacción del Sistema de Transacciones y el Intéprete del Sistema de Reglas

En el área de base de datos activas se define que una situación necesaria para evaluar las reglas es un evento. Desde el punto de vista de aplicar las capacidades activas para conseguir la imposición automática de consistencia, un evento es un requerimiento de actualización de una entidad de la base, que puede ser el causante de violar alguna restricción y, por lo tanto, determina la necesidad del chequear las reglas especificadas sobre las entidades.

Con los eventos, la situación que especifica la parte izquierda de una regla da lugar a la forma evento-condición. Como resultante tenemos un nuevo tipo de reglas, propio de las bases de datos activas: las reglas evento-condición-acción. [McCar89]

III - Reglas Evento-Condición-Acción (ECA)

Una regla ECA consiste de tres partes: evento, condición, y acción. Cuando el sistema detecta que un evento ha ocurrido, evalúa la condición y, si ésta es satisfecha, ejecuta la acción. Los eventos y las condiciones juegan diferentes roles: los eventos definen *cuándo* se debe chequear una regla, y las condiciones especifican *qué* se debe evaluar para determinar si la acción de la regla tiene que ser ejecutada [WiFi90]. El momento preciso de la evaluación de dicha condición da lugar a distintas alternativas que el DBMS puede implementar.

Se puede decir que la ocurrencia de un evento *indica* la necesidad de *evaluar* las restricciones especificadas sobre la base. Esta indicación no implica que la regla se evalúe en forma inmediata a la ocurrencia del evento. El momento preciso de dicha evaluación lo administra el sistema de transacciones, cuando le pasa el control al intérprete del SR. Existen varias posibilidades para elegir dicho modo de acoplamiento:

1 - Postergar toda introducción de tokens hasta el final de la transacción. Es claro que un paso de una transacción puede provocar uno o más eventos.

2 - Postergar, con algún criterio, la introducción de algunos tokens, y otros introducirlos en el momento que ocurre el evento.

3 - Introducir todos los tokens en el momento en que ocurre el evento.

Tanto la primera, como la última alternativa tienen como desventaja su rigidez. El hecho que siempre se evalúe una regla inmediatamente después de la ocurrencia del evento hace posible que un mismo token sea ingresado, y procesado por el intérprete, más de una vez durante la transacción. Por el otro lado, si siempre se chequean al final, y teniendo en cuenta como objetivo principal de los mecanismos activos la imposición automática de consistencia, se van a detectar las violaciones a las restricciones tardíamente. Como consecuencia, habrá pasos de la transacción que estarán trabajando sobre una base de datos inconsistente, pudiendo haberse detectado y corregido antes.

La alternativa más ventajosa es la segunda, puesto que el programador de la aplicación tiene la posibilidad de decidir en que momento debe realizarse el chequeo de las reglas, pero implica que el sistema tenga estructuras para elegir el modo de interacción.

CAPÍTULO 5
PROPUESTA PARA UN MODELO ACTIVO

En los capítulos anteriores se vio que las bases de datos activas integran las capacidades de los DBMS convencionales con las capacidades de los sistemas de reglas. También se resaltó la imposición automática de consistencia como el aporte más destacado conseguido con esta integración. Desde este punto de vista, la cuestión central de la integración es que la especificación de las restricciones pueda realizarse en el momento de la definición de las entidades. Las reglas que surgen desde estas restricciones formarán parte de la modelización de las entidades que poblarán la base y serán administradas por el DBMS.

Los puntos de estudio más importantes para incorporar sistemas de reglas en bases de datos cuando se intenta proponer un DBMS activo son: la forma de especificar reglas, el modo que interactúa el modelo de ejecución del DBMS y el intérprete del sistema de reglas, y también las cuestiones de estrategia de optimización que use este último para evaluar los cambios en la memoria de trabajo.

En este trabajo se enfatiza principalmente en como integrar razonablemente los mecanismos activos en un modelo de BDOO. Para poder analizar con mayor claridad esa integración se tomó como referencia a SIGMA, un *Modelo Unificado para Bases de Datos Orientadas a Objetos* [SmTa93]. Sin embargo, los fundamentos de la propuesta podrían ser adaptados para otros modelos.

Aquí se definen las principales características necesarias de incorporar en SIGMA para convertirlo en un modelo activo. El contenido de este capítulo está organizado de la siguiente manera:

a) Una breve descripción del modelo de datos de SIGMA

b) La definición del formalismo para especificar las restricciones y como éstas derivan en reglas.

c) El Modelo de Ejecución: arquitectura del funcionamiento de los distintos módulos del sistema de base de datos.

d) Funcionamiento del intérprete del sistema de reglas.

e) Una discusión de cómo influyen las herramientas de modelización provistas por el modelo de la BDOO, como por ejemplo la composición, en el comportamiento del sistema de reglas.

En SIGMA se propone un modelo de datos orientado a objetos en el que pueden especificarse *axiomas* sobre las entidades modelizadas, los cuales son predicados de restricción que los objetos siempre deben cumplir. Desde esos axiomas se derivarán luego las condiciones de las reglas que implementarán la imposición automática de consistencia.

I. Modelo de datos SIGMA

En SIGMA se propone una conceptualización de la base de datos en tres niveles: el plano de definición, el plano de construcción del esquema, y el plano de implementación.

En el plano de definición se especifican los tipos organizados jerárquicamente y con eventuales vínculos de composición entre ellos. En el plano de construcción del esquema se seleccionan aquellos tipos, del plano de definición, que resultan de interés para la base de datos a modelizar, se agregan vínculos y restricciones propios de la misma, y se definen relaciones entre los tipos elegidos.

El plano restante consiste en la construcción de implementaciones para los tipos del universo. Es posible definir múltiples implementaciones para cada tipo pero en tiempo de ejecución la correspondencia entre un tipo y su implementación es uno a uno.

I.1 Plano de definición

Las operaciones de un tipo se diferencian en rasgos y comportamiento. Los *rasgos* definen características descriptivas de los objetos del tipo especificado, son las operaciones que permiten observar el estado de dichos objetos. El *comportamiento* es el conjunto de actividades que los objetos del tipo especificado están capacitados para llevar a cabo.

Existen cuatro maneras de construir tipos. Un tipo puede ser: básico, extensión de un tipo existente (relación *es-un*), composición de tipos existentes (relación *parte-de*), o una colección de tipos (conjuntos, relaciones).

En la definición de un tipo se pueden especificar *axiomas* que definen restricciones las instancias del mismo. Debido a que las expresiones que representan un axioma necesitan predicar sobre el estado de un objeto, el sistema tiene que proveer una forma cómoda de especificarlo. El hecho que los rasgos de un objeto sean funciones que permiten observar el estado del mismo facilita la definición de restricciones semánticas.

I.2 Plano de construcción del esquema

Definir el esquema de la base de datos consiste en:

- Seleccionar los tipos desde el plano de definición que resultan de interés para la aplicación.
- Establecer restricciones y vínculos propios de la situación que se está modelizando.

Además pueden ajustarse las relaciones parte-de, establecidas para los tipos compuestos, y definirse objetos excepcionales no necesariamente pertenecientes a un tipo en el esquema.

Cada tipo del esquema tiene una *clase básica* asociada que agrupa todas las instancias del mismo. El modelo permite especificar axiomas en la definición de un tipo y en la construcción del esquema. Los *axiomas* son predicados que definen restricciones de manera que cada instancia de la clase básica debe satisfacerlos.

Desde los axiomas se derivan las reglas que llevarán a cabo la imposición automática de consistencia.

I.3 - Sintaxis

En el plano de definición se encuentran diferentes constructores que permiten representar: tipos, subtipos, tipos compuestos y relaciones.

```
<expr_de_def_tipo> ::= < esp_de_tipo>|< esp_de _relacion>
< esp_de_tipo>  ::=   Otype <expr>
                            Inherits: <expr>
                            Features: <expr>
                            Parts: <expr>
                            Behavior: <expr>
                            Axioms: <expr>
                      EndType
< esp_de _relacion>  ::= Relationship <expr>
                            Components: <expr>
                            Inherits: <expr>
                            Features: <expr>
                            Behavior: <expr>
                            Axioms: <expr>
                         EndType
<expr> ::= ...
```

I.3.1 Un ejemplo

Construcción de tipos: suponiendo que se intenta modelizar la situación de una Asociación de fútbol, dirigida por un grupo de autoridades en la que intervienen diversas instituciones con sus equipos y que organiza un campeonato anual. La siguiente es la construcción de un posible universo de tipos, utilizando el mismo ejemplo que [SmTa93].

```
Otype Persona
        Features:
                nombre: String;
                edad: integer; ...
        Behavior
                cumplir_años: Persona->Persona; ...
        Axioms
                edad(P) >= 0
Endtype
```

```
Otype Futbolista
        Inherits: Persona
        Features:
                puesto: String;
                cant_de_goles: Integer;
        Behavior:
                hacer_goles:Futbolista->Futbolista
        Axioms:
                cant_de_goles >= 0
Endtype
```

Un conjunto de futbolistas es una instancia genérica del tipo genérico conjunto:

Conjunto_de_Futbolistas : Conjunto(Futbolista)

Los Futbolistas junto con su entrenador conforman un equipo.

```
Otype Equipo
        Features:
                nombre: String;
        Parts
                director_técnico: Persona;
                futbolistas: Conjunto_de_Futbolistas;
        Behavior: ...
        Axioms: ...
Endtype
```

II - Restricciones en el modelo de datos

Las restricciones de integridad son propiedades que cualquier estado de la base de datos debe verificar para que la misma no sea inconsistente, con lo cual un estilo declarativo resulta apropiado para su representación. La correctitud es la característica más importante al elegir ese formalismo para tener la seguridad que puede determinarse si una condición se satisface, en función del contenido actual de las clases de la base que intervienen en un predicado [CFPT93].

El lenguaje de especificación de restricciones puede ser expresado: por una adaptación del lenguaje SQL [CeWi92], o sentencias en un 'Lenguaje lógico temporal de Primer Orden' [ECO93], también puede ser representado como una fórmula cerrada del

cálculo del dominio relacional [CFTP94], o en una representación clausal como la que usa Prolog [LiLe89].

El lenguaje de restricciones de esta propuesta está formado por predicados básicos que se traducen luego en *patrones* de las reglas. Los predicados básicos son las expresiones más sencillas del lenguaje que son interpretables como aseveraciones, como por ejemplo que un determinado objeto verifica una cierta propiedad [Ham81]. Son la materia de la que están hechos los axiomas y se pueden combinar de acuerdo a la gramática que se especifica mas abajo. Precediendo la definición de un axioma se declaran las variables y su dominio.

La gramática que describe la sintaxis de los axiomas es la siguiente:

<axioma> ::= <defVariables>.<fbf>
<defVariables> ::= <defVar> | <defVar>, <defVariables>
<defVar> ::= <var>: <nombreDeClase> | <var>,<defVar>
<fbf> ::= <patrón> | (<patrón> And <fbf>) | (<patrón> Or <fbf>) | Not(<fbf>)
<patrón> ::= <predicadosBásicos> | <predicadosAgregados>
<predicadosBásicos> ::= *Igual*(<exp>,<exp>) | *Menor*(<exp>,<exp>) |
Mayor(<exp>,<exp>) | *Pertenece*(<exp>,<exp>) | ...
<predicadosAgregados> ::= ...
<exp> ::= <valor> | <var> | <mensaje>(<exp>)
<valor> ::= <string> | <número> | <lógico> | <oid>
<mensaje> ::= ....

La fórmula que especifica un axioma se convertirá en la condición de una regla que será usada para 'chequear' si un axioma no se cumple en un momento dado. Un módulo del sistema, llamado Evaluador de Condiciones, es el encargado de resolver el valor de verdad de la fórmula cuando la regla deba ser chequeada. Este módulo es un intérprete que evalúa la condición de una regla efectuando distintas asignaciones de valores a las variables.

Como el evaluador no puede resolver la expresión <mensaje>(<exp>) interactúa con el Administrador de Objetos que le devuelve el valor que la misma representa. Cabe aclarar que el resultado de dicha expresión podría ser un objeto, pero en tal caso el

administrador devolverá el *identificador del objeto* (oid) y desde la gramática definida más arriba surge que un oid es un valor.

Para comprender la interpretación semántica intuitiva de los patrones que componen los axiomas se exponen los siguientes puntos:

1 - Las variables que se definen en la parte izquierda de un axioma, antes de la fbf, se corresponden con predicados básicos. Los valores que satisfacen uno de esos predicados, es decir que son su interpretación, son aquellos objetos que pertenecen a la clase dominio de la variable. Por ejemplo la definición de una variable X: Persona se corresponde con el predicado Persona(X), cuya interpretación es el conjunto de objetos que forman la clase Persona.

2 - Los predicados vistos en el punto 1 se relacionan entre ellos, y con la fbf, para formar la condición de la regla por medio del conectivo $\Rightarrow$ (entonces). De esta manera, el axioma

X,Y: Persona. Not(Igual(dni(X),dni(Y))) Or Igual(X,Y)

está significando que si X e Y son objetos de la clase persona entonces o bien sus dni son distintos o bien X e Y son la misma persona.

Simbólicamente quedaría:

$(\forall x)(\forall y)\ [(Persona(x) \wedge Persona(y)) \Rightarrow (\neg Igual(dni(x),dni(y)) \vee Igual(x,y))]$

3 - Cada patrón se corresponde con un predicado que tiene una interpretación semántica dada, es decir, tiene una implementación en el Evaluador de Condiciones. Son triviales las interpretaciones de los predicados básicos que aparecen en la gramática (Igual, Mayor, etc.). El modelo provee una base de patrones que el diseñador puede extender especificando predicados en el Evaluador de Condiciones, y definiéndoles la interpretación semántica correspondiente. Posteriormente éstos patrones pueden ser utilizados en la especificación de los axiomas de los tipos.

II.1 - Generación de reglas

La sola existencia de una restricción especificada en el axioma de un tipo, no alcanza para realizar, por ejemplo, la imposición automática de consistencia. Es

necesario que ese axioma se derive en una, o más reglas, que sean el soporte para la ejecución de acciones correctivas, de cancelación, o simplemente pasos en la solución de algún problema cuando el mismo deje de valer para un determinado estado de la base.

La creación de reglas necesita de un compilador, que analice que métodos son generadores de eventos, transforme los axiomas en condiciones de reglas, y genere toda la información necesaria para que el intérprete del sistema de reglas evalúe el estado de la base.

La condición de una regla representará la negación de un axioma. Por ejemplo, para un axioma dado

X,Y: Persona. Not(Igual(dni(X),dni(Y))) Or Igual(X,Y)

cuya interpretación intuitiva es

$(\forall x)(\forall y)$ [(Persona(x) $\wedge$ Persona(y)) $\Rightarrow$ ($\neg$Igual(dni(x),dni(y)) $\vee$ Igual(x,y))]

que es lógicamente equivalente a

$(\forall x)(\forall y)$ [$\neg$(Persona(x) $\wedge$ Persona(y)) $\vee$ ($\neg$Igual(dni(x),dni(y)) $\vee$ Igual(x,y))]

se deriva en la condición

$\neg${ $(\forall x)(\forall y)$ [$\neg$(Persona(x) $\wedge$ Persona(y)) $\vee$ ($\neg$Igual(dni(x),dni(y)) $\vee$ Igual(x,y))] }.

que es lógicamente equivalente a

$(\exists x)(\exists y)$ [(Persona(x) $\wedge$ Persona(y)) $\wedge$ (Igual(dni(x),dni(y)) $\wedge$ $\neg$Igual(x,y))] .

Una regla consta de una condición y una acción asociada. La acción es un mensaje aplicado a un objeto que seguramente modificará la base para poder restaurar la consistencia. Para el sistema de reglas tendrá el efecto de un cambio en la memoria de trabajo.

Puede notarse que las reglas generadas para un tipo de objetos pasan a formar parte de la representación conceptual del mismo ofreciendo una mayor capacidad de modelización semántica.

II.2 - Efecto de la herencia en las reglas

Así como los rasgos y el comportamiento de un tipo de objetos son propiedades del mismo también las reglas los son y, por lo tanto, son heredadas por todo subtipo suyo.

Para analizar como afecta la herencia en las restricciones cuando se define un subtipo de otro con algún axioma definido se propone el siguiente ejemplo:

Una base de datos donde está definido el tipo Persona con el rasgo dni y la restricción que: 'No hay dos personas con igual dni'.

```
Otype Persona
        Features
                dni : Nat;
        Axioms
                X,Y: Persona. Not Igual (dni(X),dni(Y)) Or Igual(X,Y)
EndType
```

Si ahora se define el tipo Estudiante como subtipo de Persona, el mismo hereda la restricción especificada.

Como un objeto estudiante está incluido en la clase de las personas, cuando se evalúe la condición anterior las variables se ligarán con los objetos de la clase dominio. Por lo tanto, está asegurado que los objetos estudiantes cumplen las restricciones de las personas y no habrá un estudiante con igual dni que otra persona.

III - El modelo de ejecución

En este punto se describe el funcionamiento de las distintas componentes del modelo, incluyendo las que incorporan las capacidades activas y la manera en que interactúan entre ellas. Las funciones que implícitamente se han ido descubriendo a lo largo de este capítulo como evaluar condiciones de las reglas, evaluar mensajes, etc. tendrán sus módulos ejecutores en la arquitectura del sistema. Además, aparecen las dos funciones principales que gobiernan el flujo de control de las operaciones de base de datos que solicitan las aplicaciones: administrar las transacciones, y administrar el sistema de reglas.

III.1 - Arquitectura del sistema

Las componentes funcionales del sistema son cuatro: el Sistema de Transacciones(ST), el Intérprete del Sistema de Reglas(ISR), el Administrador de Objetos, y el Evaluador de Condiciones.

El ST y el ISR comparten la administración del flujo de control durante la ejecución de una operación sobre la base de datos. Los otros dos módulos cumplen funciones específicas, utilizadas por el ST y el ISR.

Seguidamente se describen las funciones que realizan cada una de las componentes y la interacción que mantienen entre ellas:

- Sistema de Transacciones: ejecuta las operaciones sobre la base de datos solicitadas desde una aplicación y realiza la actualización de los objetos. Cada paso de una transacción es un mensaje que se corresponde con un método de algún tipo, el cual debe ser recuperado por el Administrador de Objetos.

Cuando se produce un evento el ST le pasa el control al ISR con la información del objeto modificado. Ese objeto es llamado token.

Finalmente, el ST informa a la aplicación el resultado de la transacción solicitada.

- Intérprete del Sistema de Reglas: es responsable de chequear las reglas generadas para asegurar la consistencia de la base de datos, manteniendo en su memoria de trabajo toda la información necesaria para realizarlo de un modo óptimo.

En el inicio de una transacción se supone que el estado de la base es consistente, es decir, ninguna regla se satisface. Al producirse un cambio en algún objeto de la base se genera un token que debe ser procesado por el ISR.

El ISR toma el token y lo introduce en una estructura de información propia generada para determinar las instancias de reglas que conformarán el conjunto de conflicto. Durante ese proceso interactúa con el Evaluador de Condiciones para resolver el valor de verdad de los patrones presentes en la condición.

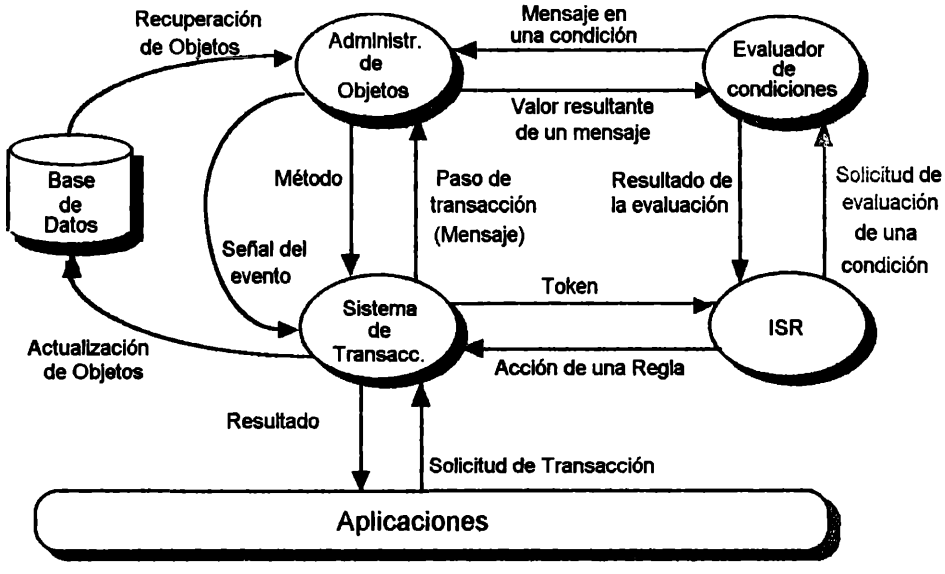
Por último, debe seleccionar una regla desde el conjunto de conflicto y solicitarle al ST la ejecución de la acción asociada.

- Administrador de Objetos: provee el manejo de los objetos. Entre otras cosas, cumple la función de recuperar el método correspondiente a un mensaje para que el ST lo ejecute. Además incluye el proceso detector de eventos.

Un evento es un mensaje que modificará el estado del objeto receptor. Los métodos que son eventos se determinan en tiempo de compilación. De esta manera, el

mecanismo de detección de eventos es tan simple como verificar si el método que se va a ejecutar está caracterizado como generador de eventos.

- Evaluador de Condiciones: evalúa la condición de un nodo haciendo las asignaciones posibles de valores a sus variables. Si en la condición está presente un mensaje tiene que interactuar con el Administrador de objetos.



Componentes funcionales del OODBMS activo

Desde la descripción de las componentes funcionales puede notarse que el flujo de control de una transacción 'alterna' entre el ST, el ISR y el Evaluador de Condiciones. En efecto, cuando se produce un evento el ST inicia una interacción con el ISR. Luego el control vuelve al ST para ejecutar la acción de la regla seleccionada. Dicha acción posiblemente modifica algún objeto de la base y, por lo tanto, genera un nuevo token. La interacción continúa hasta que no se instancie ninguna regla o se ejecute una operación de abortar la transacción. Por último, el ST retomará la ejecución en el punto donde se interrumpió.

Hasta aquí quedó claro que una vez detectada la ocurrencia de un evento el ISR debe evaluar las reglas que pudieron verse afectadas. Sin embargo, como se vio en el

capítulo cuatro, el momento en que se produce la interacción entre ambos módulos da origen a distintas alternativas de lo que se llama acoplamiento evento-condición.

III.2 Acoplamiento evento-condición

Los diferentes modos de acoplamiento definen en que momento deben evaluarse las restricciones. En algunas situaciones no se necesita, y hasta puede resultar problemático, realizar el chequeo inmediatamente después de la ocurrencia del evento. Por ejemplo, si se tiene definido el tipo Persona con un axioma que impone que: 'si una persona x tiene un cónyuge y entonces el cónyuge de y debe ser x'

```
Otype Persona
        Features
                Dni: Nat;
                Nombre: String;
        Parts
                cónyuge: Persona;
        Behavior
                AsignarDni: Persona, Nat -> Persona
                Casarse_Con: Persona, Persona -> Persona
        Axioms
                P:Persona. Igual(P,cónyuge(cónyuge(P))) Or
                                        Igual(cónyuge(P),NULO)
        Endtype
```

y una transacción que represente el casamiento de dos personas, escrita como:

```
COMIENZO_DE_TRANSACCION  Casamiento
        A.Casarse_Con: B;
        B.Casarse_Con: A;
FIN_DE_TRANSACCION Casamiento
```

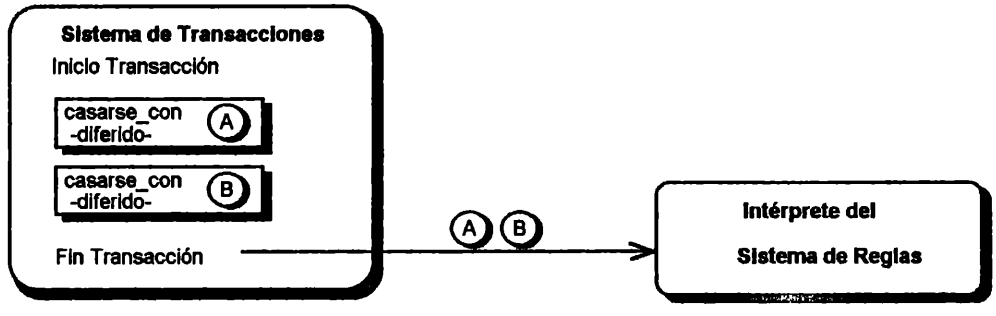
El método 'Casarse_Con' le asigna al atributo 'cónyuge' del objeto Persona que recibe el mensaje el objeto enviado como parámetro. Por lo tanto, 'Casarse_Con' origina un evento porque modifica el estado del objeto receptor.

Si la evaluación de la regla se realizara inmediatamente después a la ocurrencia del evento el ISR detectaría una violación de la restricción. Observando la transacción,

puede notarse que en el paso donde A se casa con B no existe tal violación, sino que la evaluación de la condición debería postergarse hasta que se asiente el casamiento de B con A.

Por lo tanto, se deducen dos alternativas para el momento de la introducción de un token en el ISR: *inmediato* o *diferido*. En el modo inmediato el token se introduce al finalizar el evento. En el modo diferido se introduce al finalizar la transacción.

El modo diferido tiene utilidad en situaciones en las cuales un paso de una transacción necesariamente viole una restricción que luego se corrige en un paso posterior. Por otro lado, este modo posibilita reducir la cantidad de evaluaciones de las condiciones de las reglas, provocadas por varias actualizaciones a un mismo objeto en una misma transacción. En [Han92] se definen eventos lógicos y eventos físicos para representar esta característica de chequeo de reglas.



Modo de acoplamiento Diferido

Recordando ahora que el tipo persona también tiene definido el axioma:

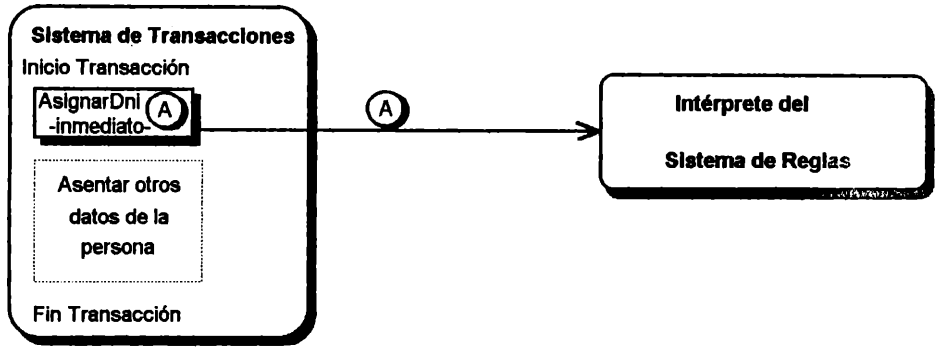
X,Y: Persona. Not Igual(dni(X),dni(Y)) Or Igual(X,Y).

y suponiendo que existe una base de datos en la cual se registrarán todas las personas habilitadas para votar, la cual es actualizada periódicamente desde la base del Registro Nacional de las Personas por un proceso que cuenta con la siguiente transacción:

```
COMIENZO_DE_TRANSACCION  RegistrarVotante
        unaPersona.AsignarDni: unDni
         // Asentar otros datos de la persona (Dirección, etc) //
FIN_DE_TRANSACCION RegistrarVotante
```

En el paso donde se asigna el DNI a la persona debe chequearse que ese número no sea el dni de otra. Teniendo en cuenta que // Asentar otros datos de la persona

(Dirección, etc) // puede implicar una serie de actualizaciones resulta más apropiado tener el control en el punto donde se asignó el dni y no al final de la transacción.



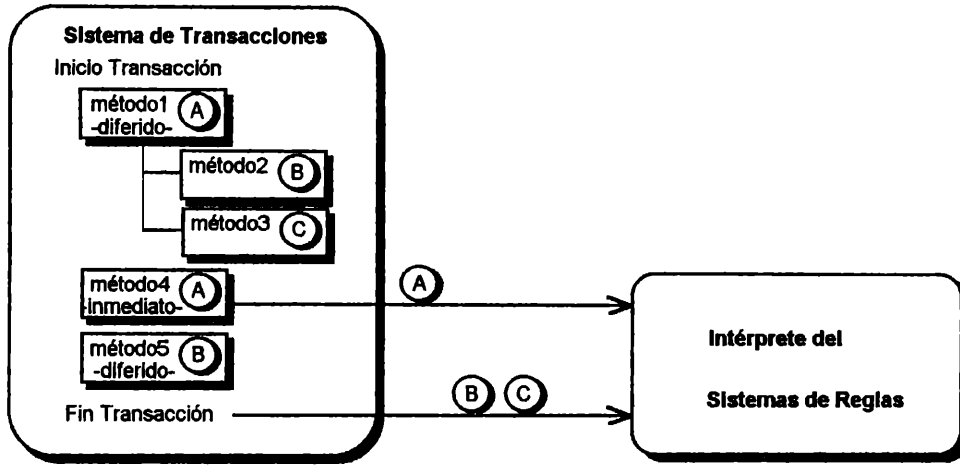
Modo inmediato

Hasta aquí se analizó el modo inmediato en su forma más simple, es decir, cuando un paso de una transacción es un evento, genera un token, y acto seguido se lo introduce en el ISR.

Sin embargo, en cada paso de una transacción puede producirse una secuencia de eventos, llamados eventos hijos, correspondientes a los mensajes que conforman la implementación del método vinculado con ese paso, llamado evento padre. Lógicamente, por cada evento hijo se genera un token. Si el evento padre fue definido de modo inmediato surge un interrogante: que sucede con todos los tokens que se generan a partir de un evento padre? Se introducen a medida que se producen eventos hijos, o al finalizar el evento padre?

Para resolver lo anterior surgen dos variantes del modo inmediato: el *inmediato-inmediato*, y el *inmediato-diferido*. En el modo inmediato-inmediato los tokens se introducen en el ISR a medida que se producen los eventos hijos, en el modo inmediato-diferido se postergan hasta el fin del paso de la transacción.

Por último, debido a que para cada paso de una transacción puede definirse un modo de acoplamiento es posible que exista una combinación de modos de acoplamiento entre los pasos de una misma transacción. Un paso elemental, definido como diferido, determina que los tokens que se generen a partir él no sean introducidos en el ISR hasta el fin de la transacción. Sin embargo, si alguno de esos tokens es generado también en un paso posterior y de modo inmediato, será introducido antes.



Combinación de modos de acoplamiento

IV- Intérprete del sistema de reglas (ISR)

El objetivo del ISR es resolver que reglas se instancian a partir de la configuración corriente de la memoria de trabajo. La memoria de trabajo contiene una referencia de cada objeto que existe en la base de datos. Cada cambio en el estado de la base produce también un cambio en la memoria de trabajo. En este contexto se habla indistintamente de base de datos y memoria de trabajo.

Desde un estado inicial, en donde ninguna condición de regla se satisface, al introducirse un token el estado de la memoria de trabajo cambia y el ISR debe evaluar la condición de cada regla con cada instanciación posible de objetos de la base. Si encuentra una instanciación que satisface la condición de una regla la incorpora en el conjunto de conflicto. Después de haber evaluado todas las reglas el ISR elige, desde el conjunto de conflicto, una instancia de alguna de ellas para ejecutar la acción asociada.

Como se vio en el capítulo 4, el intérprete debe resolver cuál es el mejor modo de determinar los cambios en el conjunto de conflicto que resultan desde los cambios en la memoria de trabajo.

El modo de optimizar el procesamiento de un cambio en la memoria de trabajo en un sistema de reglas es estructurando la misma, formando lo que se llama una *red de discriminación,* de manera de:

a) Evitar la iteración sobre la memoria de trabajo: significa no acceder a todos los objetos de la base para evaluar la condición de una regla.

b) Evitar la iteración sobre la memoria de reglas: significa no chequear todas las reglas cuando se produce un cambio en la memoria de trabajo.

IV.1 Como evitar la iteración sobre la memoria de trabajo

En [For82] se sostiene que la iteración puede ser evitada almacenando con cada patrón una lista de los objetos con los que coincide('match'). Las listas son actualizadas cada vez que la base de datos cambia: cuando un objeto entra a la memoria de trabajo el intérprete encuentra todos los patrones con los que coincide y lo agrega a sus listas; cuando un objeto sale de la memoria de trabajo el intérprete nuevamente encuentra todos los patrones con los que coincide y lo elimina de sus listas.

En el Anexo 1 se detallan las principales técnicas que usan los ISR para optimizar el procesamiento y se describen las características de los dos intérpretes más conocidos: RETE y TREAT.

IV.2 Como evitar la iteración sobre la memoria de reglas

Para evitar la iteración que requiere la evaluación una a una de las reglas Rete[For82] y Treat[Mira90] usan una red de discriminación, generada desde la compilación de los patrones que forman las condiciones.

El procesamiento en la red comienza introduciendo un token en el nodo raíz, obteniéndose como resultado el conjunto de conflicto formado por las instancias de las reglas cuya condición se satisface.

Cada nodo de la red representa un patrón de la condición de una o más reglas. En el ISR que aquí se propone los patrones de un predicado pueden clasificarse como:

a) Intra-objeto: cuando involucra un único objeto.
Ejemplo: Otype Votante
Inherits: Persona, Dirección;
vota_en: Dirección;
...
Axioms
X: Votante. Edad(X) >= 18;
EndType

b) Inter-objeto: cuando involucra a más de un objeto. Existen dos variantes:

b.1) Intra-clase: definen una relación entre objetos de una misma clase.

Ejemplo: Otype Persona
Features
dni : Nat;
Axioms
X,Y: Persona. Not Igual (dni(X),dni(Y)) Or Igual(X,Y)
EndType

b.2) Inter-clase: cuando definen una relación entre objetos de distintas clases.

Ejemplo con una restricción definida en la construcción de un esquema:

En el plano de definición de tipos están definidos:

Otype Persona	Otype Animal
Features	Features
nombre: string;	nombre: string;
EndType	EndType

En el plano de construcción del esquema se define el siguiente axioma:

X: Persona, Y: Animal. Not Igual (nombre(X),nombre(Y))

Asociado a los nodos de la red se mantiene información de los tokens que satisfacen un patrón para reducir el costo que implica la exploración de la memoria de trabajo, cuando se evalúan las condiciones de las reglas.

En Rete todo el estado de la memoria de trabajo está en la red de discriminación almacenada en los nodos inter-objeto (beta-nodos). Cada beta-nodo (ver Anexo 1) tiene dos listas, una con los tokens que ingresaron por cada una de las clases que intervienen en la relación que representa. Los beta-nodos también son llamados nodos de doble entrada ya que representan un 'join' entre dos listas de objetos, provenientes dos nodos anteriores en la red.

Treat, en cambio, mantiene los tokens en los nodos intra-objeto y también las instancias que forman el conjunto de conflicto. Los beta-nodos son condiciones de 'join' que unifican dos nodos anteriores en la red, y son evaluados durante el procesamiento.

El ISR a incorporar en la arquitectura que se está proponiendo adoptará la estrategia de Treat. La red de discriminación tendrá dos sentidos, uno para representar el grafo de tipos con el que se formó el esquema de la base de datos, el otro para las condiciones de las reglas que surgen desde los axiomas de los tipos de la base. De esta manera, la red presentará las siguientes características:

1- Una capa que determina en cuáles subredes debe introducirse un token, correspondiente al grafo de tipos.

2 - Las clases de objetos correspondientes a los tipos, las cuales abren una subred para cada uno de ellos en el sentido de las condiciones de reglas. Tienen una correspondencia con los alfa-nodos (ver Anexo 1) de Treat porque mantienen una referencia a las instancias de los objetos.

3 - Una capa de alfa-nodos que filtran las condiciones intra-objeto de las reglas de un tipo. Consiste de nodos virtuales ya que no tienen una lista asociada para mantener los objetos que satisfacen la parte de condición que representan, sino que sólo la especifican para que sean recuperados durante el procesamiento de un token por el Evaluador de Condiciones.

Eventualmente podrían tener la lista de objetos que la satisfacen si esta no fuera demasiado grande, y así optimizar el tiempo de evaluación.

4 - Una capa de beta-nodos. Igual que en el punto 3, son nodos virtuales que representan las condiciones inter-objeto de las reglas de un tipo. Tienen una entrada por cada variable presente en esa parte de la condición.

5 - Por último, el conjunto de conflicto que contiene las instancias de las reglas cuyas acciones deben ser ejecutadas. Una instancia de regla es una tupla que representa una asignación de valores a las variables que satisfacen la condición.

IV.3 - Ejemplo de una red de discriminación

Se supone un esquema de una base de datos donde existen los tipos: Persona, Estudiante y Animal, con Estudiante como subtipo de Persona, y con los siguientes axiomas:

a) X,Y: Persona. Not(Igual (dni(X),dni(Y))) Or Igual(X,Y)

b) P: Persona. Igual(P,cónyuge(cónyuge(P))) Or Igual(cónyuge(P),NULO)

c) X,Y: Estudiante. Not(Igual (nroAl(X), nroAl(Y))) Or Igual(X,Y)

d) X: Persona, Y: Animal. Not(Igual(nombre(X),nombre(Y)))

por lo visto en *Generación de reglas* se transforman en las siguientes condiciones de reglas:

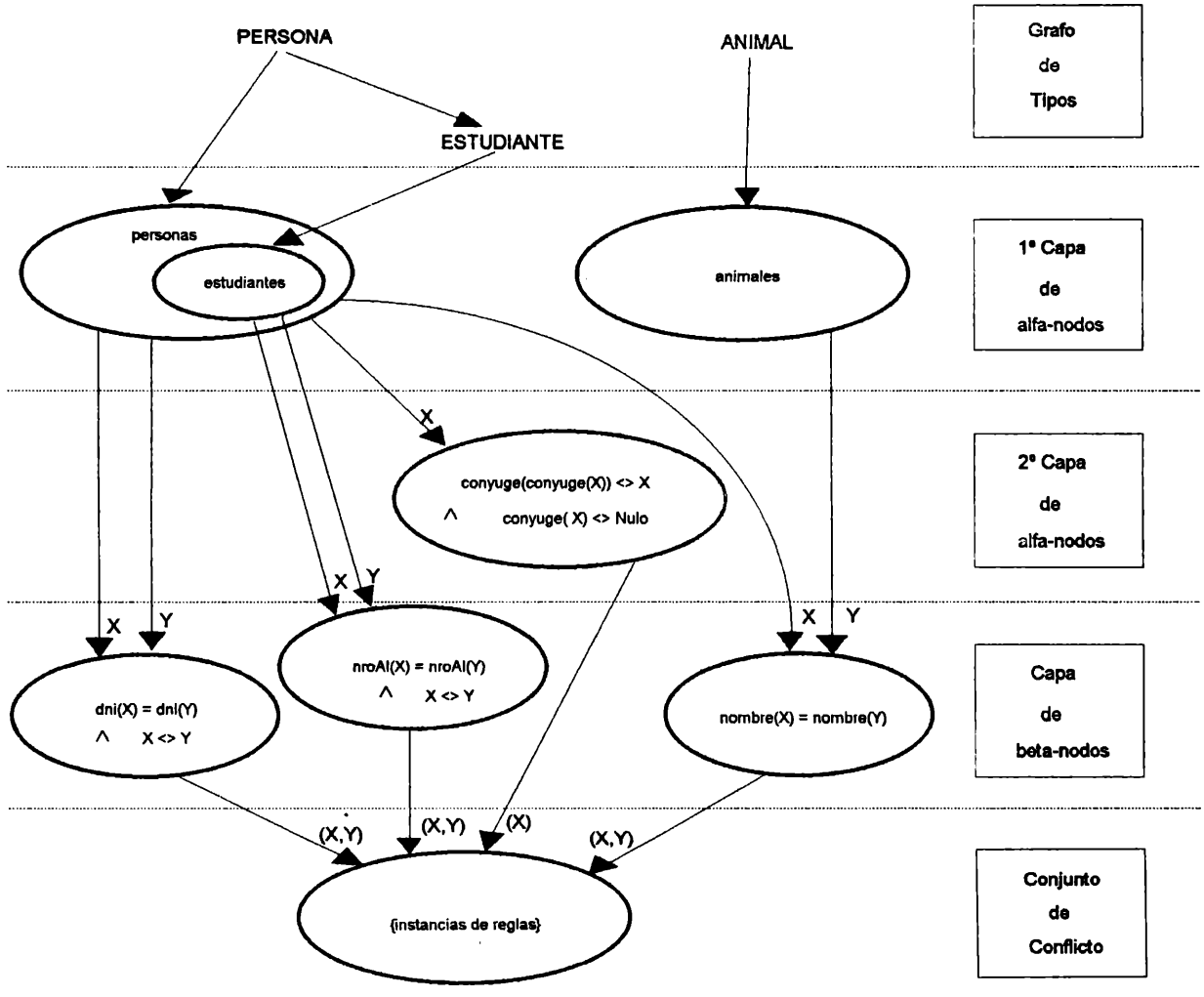
a) (∃x)(∃y) [(Persona(x) ∧ Persona(y)) ∧ (Igual(dni(x),dni(y)) ∧ ¬Igual(x,y))]

b) (∃p) [Persona(p) ∧ (¬Igual(p,cónyuge(cónyuge(p))) ∧ ¬Igual(cónyuge(p),NULO))]

c) (∃x)(∃y) [(Estudiante(x) ∧ Estudiante(y)) ∧ (Igual(nroAl(x), nroAl(y)) ∧ ¬Igual(x,y))]

d) (∃x)(∃y) [(Persona(x) ∧ Animal(y)) ∧ igual(nombre(x),nombre(y))]

Los predicados Persona, Estudiante y Animal que se corresponden con nombres de clases están ubicados en la primer capa de alfa-nodos, y puede notarse que son compartidos por más de una condición de regla. El resto de la condición se ubica en la segunda capa de alfa-nodos si solamente aparece una variable, o en la capa de beta-nodos si aparece más de una variable.



V. Influencia del modelo de BDOO en el chequeo de reglas

Se puede afirmar que las restricciones definidas para un tipo pueden ser violadas sólo por los métodos propios o heredados porque no hay otra forma de modificar el estado de un objeto. Sin embargo, esta afirmación no es completa si no se analiza que sucede con las restricciones asociadas a los objetos de un tipo T, cuando ocurre una actualización sobre un objeto de un tipo S. Es necesario tener en cuenta sólo dos situaciones en las cuales una o más de dichas restricciones pueden ser violadas.

Primero, si existe un predicado definido en el esquema de la base de datos donde aparece alguna variable ligada a una clase de tipo T y otra variable ligada a una clase de tipo S. Este predicado está representando una relación conceptual entre los objetos de ambas clases.

Segundo, si existe alguna parte del tipo T definida como del tipo S. Aquí el problema no surge de un predicado sino que es la consecuencia de una relación de composición entre ambos tipos.

A continuación se muestran ambas situaciones en dos ejemplos, y posteriormente se analizan las diferencias.

Suponiendo tener los tipos Persona y Animal, que responden al mensaje 'nombre', y un predicado de restricción especificado en el esquema de la base de datos que impone que 'no haya una persona y un animal con el mismo nombre'

```
Axioma (*de la base de datos*)
        X: Persona, Y: Animal. Not(Igual(nombre(X),nombre(Y)))
Otype Persona                   Otype Animal
    Features                        Features
        nombre: string                  nombre: string
EndType                         EndType
```

cuando se ejecute una operación que modifique el nombre de un objeto 'animal' siempre habrá que evaluar el predicado de la restricción, contra todos los objetos de la clase Persona. Del mismo modo, cuando ocurra una operación que modifique el nombre de un objeto 'persona' siempre habrá que evaluar el predicado de la restricción, contra todos los objetos de la clase Animal.

En este ejemplo no hay un vínculo de composición entre los tipos sino sólo una relación a través de un predicado.

Si ahora estudiamos el ejemplo para modelizar una restricción donde se imponga que 'los integrantes de un equipo juvenil no tengan más de 18 años',

```
Otype Persona;
    Features
        edad: Integer;
    Behavior
        culplir_Años: Persona->Persona
    EndType

Otype Jugador;
    Inherits Persona
    Features
        puesto: String;
EndType

Otype EquipoJuvenil;
    Parts
        jugadores: Conjunto(Jugador);
    Axioma
        X: Equipo, Y: Jugador. Pertenece(Y,jugadores(X)) And edad(Y)<18
EndType
```

Cuando se ejecute una operación que modifique la edad de un objeto 'jugador' habrá que evaluar la restricción de la clase EquipoJuvenil, pero sólo para aquellos objetos que sean *dueños* del objeto receptor del evento, es decir, lo tengan instanciado como una parte.

A partir de los ejemplos pueden analizarse las diferencias entre las posibles violaciones de una regla generadas por:

a) la existencia de una restricción *inter-clase* entre un tipo T y uno S,

b) y las que surgen por la composición de objetos.

a) En una restricción inter-clase, los objetos involucrados son todos los de ambas clases. Esto significa que un evento que modifique un objeto de una de ellas disparará la evaluación del predicado, iniciando una búsqueda por instanciaciones de la condición de la regla en los objetos de la otra clase.

b) Ante cualquier evento que modifique un objeto de tipo S deben evaluarse las restricciones de los objetos dueños del receptor del mensaje. Esto debe hacerse recursivamente para los dueños de los dueños del objeto receptor.

Para realizar esta evaluación el sistema provee un mecanismo llamado 'efecto colateral' de un evento. Esto describe una nueva función que debe realizar el Administrador de Objetos: obtener los objetos compositores del receptor del evento.

Como conclusión puede observarse que la ocurrencia de un evento puede generar más de un token para introducir en el ISR, uno para cada compositor del objeto receptor.

Los tokens que se generan por el efecto lateral de un evento dependen del receptor y, por lo tanto, pueden variar para dos objetos distintos de una misma clase. El momento de generación de estos tokens será determinado por el modo de acoplamiento, definido en el paso de la transacción al que corresponde el evento.

Pero, ¿qué sucede con las restricciones de los componentes de un objeto modificado? Es claro que las restricciones de un objeto componente no pueden ser violadas por una modificación al objeto compositor, a menos que haya entre ellos un predicado inter-clase, pero ese es justamente el caso anterior, y no una relación de composición.

Siguiendo el ejemplo del jugador juvenil, en una situación donde exista en la base un jugador 'x' que juega en el equipo 'A', cuando se actualice la edad de 'x' se generará un token correspondiente a 'x', con el cual se realiza el chequeo de las reglas propias de los jugadores y, por el efecto colateral, se generará otro token correspondiente al equipo donde juega.

VI - Conclusión

Un sistema de reglas usado para monitorear el estado de la base de datos y tomar acciones en forma automática e independiente de lo solicitado por una aplicación convierte al DBMS en un DBMS activo.

En este capítulo se analizaron las cuestiones relevantes para convertir el modelo de base de datos teórico SIGMA en un modelo activo. Las características definidas en este modelo, que soporta la especificación de axiomas en la definición de los tipos, con el agregado de un mecanismo de derivación de reglas desde ellos, y un intérprete de sistemas de reglas, permite extender las capacidades de modelización incorporando un adecuado soporte para la imposición automática de consistencia.

CAPÍTULO 6
PROPUESTAS EXISTENTES DE DBMSS CON CAPACIDADES ACTIVAS

Introducción

La idea de incorporar capacidades activas en los manejadores de bases de datos ha motivado la formación de varios grupos de investigación, que han desarrollado una importante cantidad de propuestas a partir las cuales pueden apreciarse distintos criterios, apoyados en una idea general: un mecanismo de reglas trabajando dentro del DBMS.

Seguidamente se mostrarán las propuestas más representativas, y se analizarán las diferencias y similitudes con respecto a la presentada en el capítulo anterior.

I - Proyecto HiPAC [McCar89]

Es un trabajo sobre base de datos que propone el modelo de reglas ECA como un mecanismo para soportar las capacidades activas.

Además, desarrolla un modelo de ejecución en el cual se especifica como se procesan las reglas, en el contexto de transacciones que se realizan sobre la BD.

I.1 - Componentes Funcionales de HiPAC

La arquitectura de HiPAC cuenta con las siguientes componentes funcionales:

Administrador de Objetos: provee el manejo de objetos. Durante la ejecución de operaciones de BD actúa como un detector de eventos, reportando las operaciones al Administrador de Reglas.

Administrador de Transacciones: provee el manejo de transacciones anidadas. Este modelo se caracteriza por tener dos tipos de transacciones: las de alto nivel, y las anidadas o subtransacciones. Una transacción anidada es aquella que está contenida dentro de otra, llamada transacción padre. Esta última puede ser de alto nivel, es decir que no está contenida en ninguna otra, o también puede ser anidada. Una transacción puede tener varias subtransacciones. Una transacción padre es completada (commited) sólo si sus subtransacciones lo fueron.

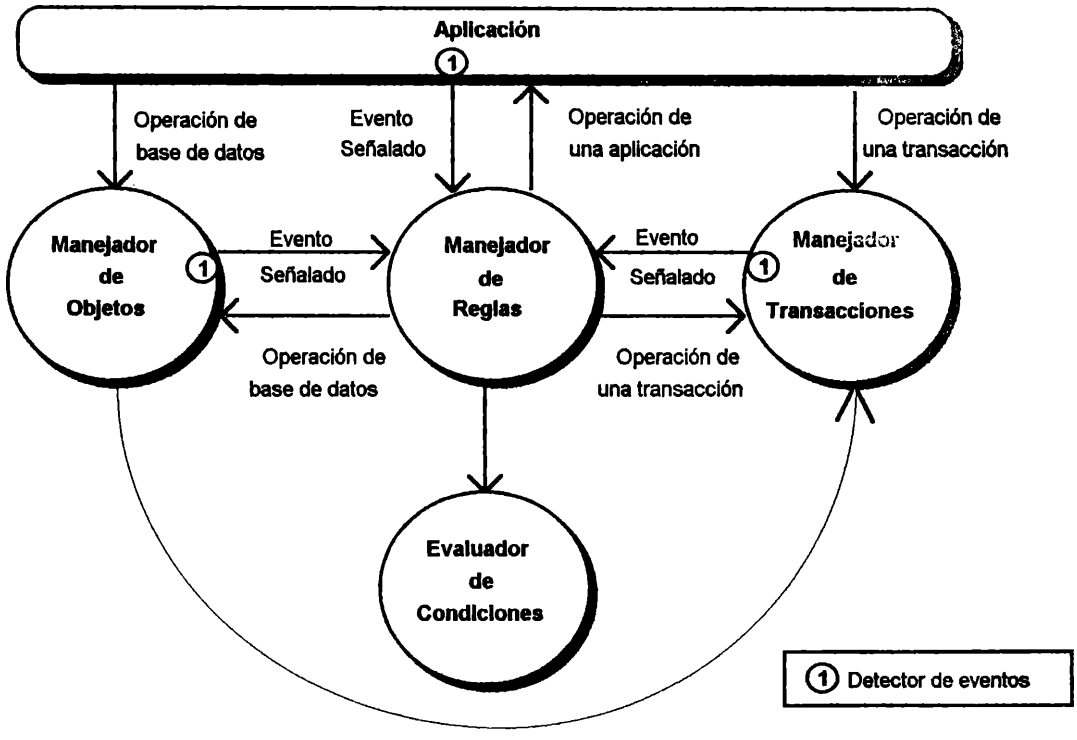
El Administrador de Transacciones actúa como Detector de Eventos, reportando la finalización de una transacción al Administrador de Reglas.

Detector de Eventos: detecta eventos primitivos y los señala al Administrador de Reglas. Hay detectores de eventos de base de datos (en el Administrador de objetos y en el Manejador de Transacciones) y para eventos de las aplicaciones (dentro de las aplicaciones). Cuando una regla es creada, el detector de eventos es programado para detectar y reportar los eventos primitivos que pueden disparar la regla.

Administrador de Reglas: mapea eventos a reglas, y reglas a transacciones. Es responsable de disparar las reglas apropiadas cuando un evento es detectado. Llama al Administrador de Transacciones para crear la transacción usada para evaluar la condición y ejecutar la acción, y llama al Evaluador de Condiciones para determinar que condiciones se satisfacen.

Evaluador de condiciones: evalúa las condiciones de reglas.

El Administrador de objetos y el Administrador de transacciones proveen la funcionalidad de un OODBMS. Las reglas ECA son implementadas por los otros tres módulos.



Componentes funcionales de HiPAC

Las componentes funcionales de HiPAC describen una configuración básica en la arquitectura de un OODBMS con capacidades activas.

I.2 - Reglas HiPAC

Las reglas son consideradas como objetos de base de datos, ellas están sujetas a las mismas operaciones que los objetos definidos por los usuarios (más algunas operaciones especiales).

Los atributos de una regla son: evento, condición, acción, acoplamiento E-C y acoplamiento C-A.

La especificación de un evento puede ser omitida en la definición de una regla, en tal caso, HiPAC lo deduce de la especificación de la condición.

La condición es una colección de queries expresados en un DML orientado a objetos que son evaluados cuando la regla es señalada por su evento.

La acción es una secuencia de operaciones de DB, o de requerimientos externos, a programas de una aplicación.

El acoplamiento E-C es la relación, relativa a los límites de la transacción, entre el momento en que es disparado el evento y el momento de la evaluación de la condición. Hay tres acoplamientos posibles: inmediato, separado y diferido. Los mismos modos de acoplamiento están disponibles para el acoplamiento C-A.

Ejemplo

Event: Update Xerox stock
Condition: where stock < 50
Action: Emitir orden de compra al proveedor A
Coupling: Condición y acción juntas en una transacción separada.

I.3 - Modelo de ejecución

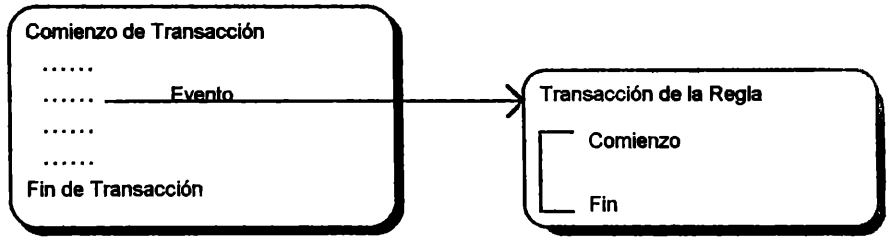
Una regla se dispara cuando su evento ocurre, entonces se evalúa la condición y, si la misma se satisface, se ejecuta la acción. Las restricciones de integridad, restricciones de acceso, datos derivados, alertadores, y otras características de las bases de datos activas son expresadas como reglas ECA.

Cuando una regla se dispara, se crea una nueva transacción.

1 - Si el modo de acoplamiento entre el evento y la condición es 'inmediato' o 'diferido', esta transacción es una subtransacción de la que contiene el evento.

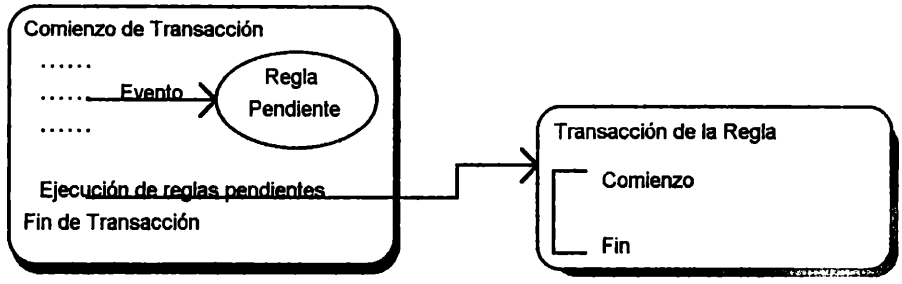
La transacción padre se suspende mientras la subtransacción es ejecutada.

a) Si el acoplamiento E-C es inmediato, esta subtransacción (de la regla) es creada y ejecutada justo en el punto donde se produjo el evento.



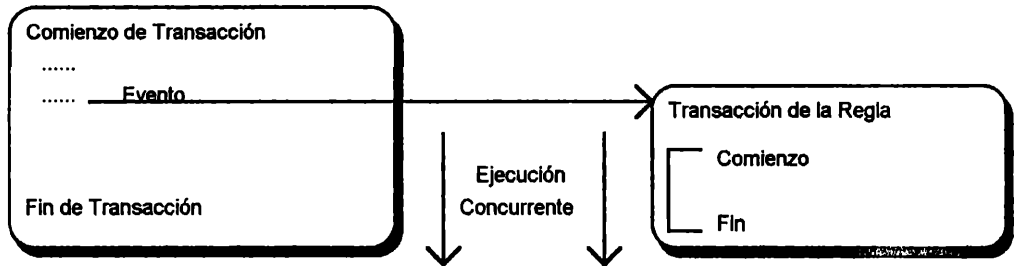
Acoplamiento E-C inmediato

b) Si el acoplamiento E-C es diferido, la subtransacción es creada y ejecutada justo antes que la transacción padre finalice.



Acoplamiento E-C Diferido

2 - Cuando el modo de acoplamiento entre el evento y la condición es 'separado', la transacción de la regla está separada de la transacción que contiene el evento, y es ejecutada en modo concurrente.



Acoplamiento E-C Separado

Si la condición de la regla se satisface, entonces se crea una nueva transacción para la acción. Su creación y ejecución dependerá del modo de acoplamiento C-A, utilizando el mismo criterio que en el acoplamiento E-C.

I.4 - Optimización

Cuando un evento dispara más de una regla, se genera una transacción para cada una de ellas. Para reglas con igual modo de acoplamiento, las transacciones se ejecutan concurrentemente. Así, no hay política de resolución de conflicto que elige una única regla para disparar, o un orden secuencial para ejecutarlas a todas. El criterio de correctitud es seriabilizado, y es impuesto por el *Administrador de transacciones*.

I.5 Comparaciones y conclusiones

En HiPAC las reglas no forman parte de la definición de las entidades sino que se usan para incrementar las capacidades de las aplicaciones. Además, en Hipac las reglas son objetos que se crean en forma dinámica. En la Propuesta del Capítulo 5 (PC5) las restricciones forman parte de la definición de las entidades y las reglas correspondientes son creadas estáticamente, pasando a conformar la modelización de las entidades.

En PC5 el modo de acoplamiento E-C puede ser diferido, inmediato-inmediato o inmediato-diferido, en cambio en Hipac existen los modos inmediato, diferido y separado. En el acoplamiento C-A en Hipac existen las mismas variantes que en el acoplamiento E-C, en PC5 solo se utiliza un acoplamiento C-A inmediato, que como se vio en PC5 resulta suficiente para realizar el control de automático de restricciones.

Del modelo de ejecución se observó que el modo de acoplamiento separado (E-C o C-A) no es adecuado para utilizarlo en las reglas que controlan las restricciones, ya que al ser la transacción de la regla independiente su ejecución no tiene efecto en la transacción que le dio origen.

HiPAC realiza una optimización importante al ejecutar todas las reglas activas de un modo concurrente. Sin embargo, la ejecución de la acción de una de ellas podría hacer desaparecer la incorrectitud que activó a varias de las otras. Se puede notar que no usa una estrategia de resolución de conflicto.

II - Facilidades de bases de datos activas en ODE [GeJa93]

Ode es una base de datos orientada a objetos que provee dos formas de soportar facilidades activas:

a) restricciones para mantener la integridad de la BD,

b) triggers para ejecutar acciones automáticamente dependientes del estado de la base.

Las restricciones aseguran la consistencia del estado de la BD, mientras que los triggers están pensados para realizar acciones específicas, cuando ciertas condiciones se vuelven verdaderas.

II.1 - Manejo de restricciones en ODE

Las restricciones son condiciones asociadas con la definición de las clases, y constan de dos partes: un predicado y una acción. La acción es ejecutada cuando la condición es falsa.

El chequeo de restricciones puede realizarse inmediatamente al acceso del objeto, o en forma diferida. Esto permite violaciones temporarias de restricciones que serán corregidas antes del 'cometido' de la transacción. Esta diferenciación da origen a dos tipos de restricciones: restricciones hard y restricciones soft.

Las restricciones hard son chequeadas inmediatamente al fin de una función propia de la clase. Si una restricción es violada, se ejecuta la acción. Luego se vuelve a evaluar la condición y si de nuevo no es satisfecha, la transacción es abortada.

Las restricciones soft se chequean recién al fin de la transacción en donde está contenida la operación que accedió al objeto. Cuando múltiples objetos están involucrados en una restricción, usualmente no es factible que la misma se satisfaga inmediatamente después de la actualización de cada objeto. Para tales situaciones es adecuado un mecanismo de chequeo de restricciones a nivel de transacción, o diferido.

La ventaja de este mecanismo se puede apreciar en el siguiente ejemplo:

```
class persona
{       ...
        persistent persona *cónyuge;
public:
        ...
soft constrain:
        (cónyuge == Null) || (this == cónyuge -> cónyuge);
};
```

La restricción anterior dice que si una persona 'x' tiene un cónyuge 'y' , entonces el cónyuge de 'y' debe ser 'x'. Si esta restricción se define como hard, nunca sería posible registrar un matrimonio, o un divorcio, porque al actualizar el primero de los dos objetos se violaría temporariamente la restricción, y causaría que cancele la transacción.

II.2 - Restricciones y triggers como reglas

En ODE no se mencionan 'las reglas' pero de hecho existen. Las clases de facilidades activas tienen forma de regla.

Los triggers también monitorean la base de datos por algunas condiciones, pero aquellas no representan violaciones de consistencia. Estos son especificados en la definición de la clase, y consisten en: un predicado de evento y una acción. La activación de un trigger debe ser hecha explícitamente para cada objeto en particular, después que el mismo ha sido creado. De esta manera, distintos objetos de la misma clase pueden tener asociados diferentes triggers. Un trigger activo se dispara cuando su predicado se vuelve verdadero, como resultado de una actualización del objeto.

Las reglas que representan los triggers son similares a las que especifican las restricciones, salvo que su condición es una expresión de evento sobre la base de datos. Las expresiones de evento son formadas usando operaciones de BD tales como la actualización de un objeto, o el 'cometido' de una transacción, operadores de composición de evento y expresiones 'comodines'.

II.3 - Optimización: Como evitar loops de reglas

Es conocido que uno de los problemas que tienen que resolver los sistemas de reglas es como evitar loops en la activación de reglas. Ode usa una estrategia simple que consiste en: volver a evaluar la condición de la regla recientemente aplicada, y si de nuevo no se satisface, abortar la activación de reglas, y por ende la transacción.

II.4 Comparaciones y conclusiones

En ODE como en PC5 las restricciones forman parte de la definición de las entidades.

Las restricciones Hard y Soft se corresponden con los acoplamientos 'inmediato' y 'diferido' de PC5. Sin embargo existe una diferencia de criterios con ODE donde el carácter de diferido se determina para cada restricción y es válido en toda transacción, en cambio en PC5 el carácter de inmediato o diferido, puede definirse en cada paso de

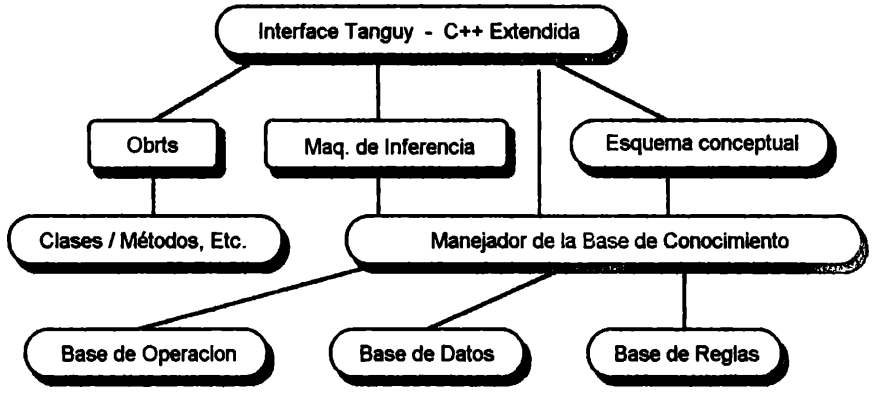
una transacción, y se aplica a todas las restricciones que estén involucradas según los objetos actualizados.

La idea básica de como realizar el control automático de consistencia en ODE es similar a la propuesta de este trabajo, en donde una operación que accede a un objeto es la causa de una posible violación de alguna restricción de la base y, por lo tanto, determina la necesidad de efectuar el chequeo de las restricciones asociadas al objeto.

III - Tanguy: Sistema manejador de bases de conocimiento [CET93]

El sistema Tanguy integra los paradigmas basados en reglas, de bases de datos, y orientados a objetos. Es una extensión de c++ que incorpora reglas de producción, almacenamiento persistente de objetos, y una interface orientada a conjuntos para la manipulación de objetos y la especificación de reglas.

Tanguy posee una 'base de conocimiento', término que proviene del área de sistemas expertos. En este caso, el objetivo de incorporar un mecanismo de reglas es poder utilizar la capacidad de inferencia del mismo. El conocimiento está expresado en las reglas, y ellas son parte de la base de conocimiento, que además está formada por los objetos, contenidos en la base de datos, y sus operaciones (base de operaciones).



Arquitectura Tanguy

III.1 - Estructura de las reglas

Las reglas Tanguy utilizan una sintaxis de C++ extendida, y su estructura es la siguiente:

RULE <nombre de la regla>

[MATCH-VARIABLES: <definición de variables C++>]

CONTEXT: <cláusula de contexto>

CONDITION: [<mensajes booleanos C++>]

ACTION: [<secuencia de mensajes C++>]

RULE-TYPE: [<before | after>]

[CONTINUE] | RETURN [<resultado>]

[PRIORITY: <número entero>]

Tanguy dispara las reglas en un contexto particular, definido en la cláusula CONTEXT donde se enumeran los mensajes que disparan la regla. En tiempo de ejecución, si la cláusula contexto de una regla machea el mensaje corriente, la regla es disparada. De otro modo es excluida del proceso de inferencia para ese mensaje.

Si bien Tanguy no usa la terminología de eventos para definir el momento en que se activará la máquina de inferencia, para analizar si alguna regla debe ser ejecutada, conceptualmente existe. Todo mensaje que machea la cláusula contexto de alguna regla es un evento.

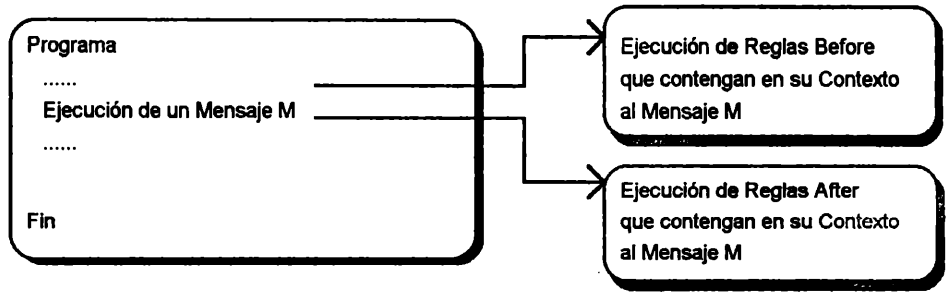
III.2 - Modelo de Ejecución

La presencia del sistema de reglas afecta el comportamiento del sistema, desviando el flujo normal de un programa hacia las reglas, cuando es ejecutado un mensaje que machea con la cláusula contexto de alguna de ellas. Básicamente por cada paso de ejecución el sistema debe:

1 - computar el conjunto de reglas 'before' que contienen en su cláusula contexto el mensaje corriente a ejecutar, luego arranca un proceso de inferencia con encadenamiento haca adelante (ver Capitulo tres) para dicho conjunto. Las reglas definidas de tipo 'before' deben ejecutarse antes del mensaje que disparó la ejecución de la regla,

2 - posteriormente, si el proceso de inferencia terminó normalmente, el programa retoma el flujo de control en el lugar correspondiente, ejecutando el mensaje.

3 - Por último, se computa el conjunto de reglas 'after' que contienen en su cláusula de contexto el mensaje ejecutado y arranca nuevamente un proceso de inferencia con encadenamiento hacia adelante para ese conjunto.



Modelo de Ejecución

III.3 Comparaciones y conclusiones

Los eventos en Tanguy se especifican explícitamente en las reglas. Este enfoque resulta demasiado rígido para resolver el problema de la 'imposición automática de consistencia'. El diseñador debe crear una regla por cada restricción que desee controlar, luego tiene que analizar que operaciones pueden causar la violación de esas restricciones, y especificarlas en las cláusulas contexto de las reglas. Si un método, que puede violar una restricción, no es tenido en cuenta en la cláusula contexto de la regla correspondiente la base de datos puede caer en un estado inconsistente sin ser detectado. Además, cada vez que se incorpora o elimina un método el diseñador debe modificar las cláusulas contexto de las reglas.

En Tanguy el mecanismo de reglas actúa en cada paso la de ejecución, y además las reglas se ejecutan en forma inmediata a la ocurrencia del evento. En PC5, se utilizan los modos acoplamiento inmediato y diferido que, como se vio anteriormente, resultan necesarios para poder realizar un chequeo de restricciones en forma flexible.

IV - Ariel : Un RDBMS con un sistema de reglas integrado [Han89] [Han92]

IV.1 - Lenguaje de reglas

El lenguaje de reglas Ariel (LRA), es un lenguaje para un sistema de reglas diseñado en un ambiente de bases de datos relacional. Las reglas Ariel pueden ser

usadas para construir aplicaciones de sistemas basados en conocimiento, también mantener restricciones de integridad de la base de datos, y monitorear estados de la misma. Las condiciones de las reglas de Ariel se basan en una combinación de eventos y macheo de patrones. También tienen un testeo de *condiciones de transición*. Las condiciones de transición tienen un valor de verdad que es una función de un nuevo estado de la BD introducido por una transacción, y un estado previo.

IV.2 - Sintaxis de las reglas

Forma general :

define rule Nombre_regla
[priority valor_prioridad **]**
[on evento **]**
[if condición **]**
then acción

La cláusula **on** permite especificar un evento que disparará la regla. Los eventos de Ariel son:

append [to] nombre_de_la_relación
delete [to] nombre_de_la_relación
replace [to] nombre_de_la_relación [(Lista de atributos)]
retrieve [to] nombre_de_la_relación [(Lista de atributos)]

IV.3 - Semántica de reglas

En Ariel los comandos son llamados de "Alto nivel" si ellos no están anidados en un comando compuesto. La ejecución de las reglas se realiza después de cada comando de "Alto nivel". Las reglas disparadas por una transacción de una BD son ejecutadas como parte de la misma.

Cláusula presente	**Condición de activación de la regla**
ON	Cuando ocurre el evento especificado después de la cláusula On.
IF	Cuando una transacción genera nuevas tuplas que machean la condición.
ON e IF	Cuando ocurre el evento y la condición se satisface.

El cuadro anterior resume cuando se activan las reglas de acuerdo a las cláusulas que la definen.

El sistema de reglas Ariel usa el modelo de reglas de producción, donde la memoria de trabajo es almacenada en la base de datos. El sistema de transacciones activa el sistema de reglas en cada *transición de base de datos*. Una transición en Ariel se define como los cambios inducidos desde una secuencia de comandos simples encerrados en un bloque *do...end*. En una transacción puede haber más de una transición, las cuales no pueden anidarse.

En Ariel, la activación de reglas se basa en eventos lógicos más que en eventos físicos. Los eventos lógicos se definen como sigue:

Tipo de actualización	Descripción	Evento lógico
im*	inserción de t seguida por cero o más modificaciones	inserción
im*d	inserción de t seguida por cero o más modificaciones, y luego borrado	nada
m+	t existía al comienzo de la transición y fue modificada una o más veces	modificación
m*d	t existía al comienzo de la transición y fue modificada cero o más veces, y luego borrada	borrado

IV.4 - Optimización

Ariel utiliza el algoritmo de TREAT para soportar el sistema de reglas. La idea de eventos lógicos y físicos se usa para reducir la cantidad de evaluaciones que el intérprete del sistema de producción debe hacer, resumiendo operaciones reiterativas sobre una misma tupla, en un único evento. Por otro lado, posibilita la tolerancia de una violación momentánea de alguna restricción, sin ejecutar una regla de restauración de la consistencia dentro de un programa que terminará dejando la base en un estado válido. Esta característica tiene utilidad en situaciones en las cuales un paso de una transacción necesariamente viole una restricción, que luego se corrige en un paso posterior.

Por ejemplo, se tiene una relación que representa a empleados, y una regla definida sobre el comando append.

empleados(nombre, nro_emple)

```
define rule NombreNoNulo on append empleados
        if empleados.nombre = "" then delete empleados
```

Si en la siguiente transacción se utilizan eventos físicos, ella no se completaría.

```
Do
        append empleados(nombre="", nro_emple = 1 )
        replace empleados(nombre="Juan")
                where empleados.nro_emple = 1
End
```

En cambio, si se utilizan eventos lógicos, según el cuadro anterior, correspondería un evento lógico de inserción, y la transacción sería llevada a cabo con éxito.

V - Cuadro comparativo

En el siguiente cuadro se enumeran diferentes puntos de análisis sobre las propuestas estudiadas, y se describe brevemente su tratamiento en cada una de ellas.

DBMS Activo	HiPAC	ODE	Tanguy	Ariel	Esta propuesta
Lenguaje de especificación de condiciones	Colección de queries expresados en un DML Orientado a Objetos	Expresión lógica C++	Expresión lógica C++	Expresiones en un lenguaje SQL	Expresiones Logicas
Reglas dentro del modelo de datos	Son Objetos y están en su propia clase.	Son parte de la definición de una clase (equiv. a tipos)	Son parte de una base de reglas	Son parte de una base de reglas	Son parte de la definición de los tipos.
Modos de Acoplamiento	Inmediato Diferido Separado	Inmediato Diferido	Inmediato	Inmediato: Al fin de cada paso de transacción, o sentencia compuesta.	Inmediato- Inmediato Inmediato- Diferido Diferido
Herencia de reglas	-	SI	No	No	SI
Efecto de la composición en la activación de reglas	No menciona ningún mecanismo	No soporta composición	No soporta composición	No soporta composición es un DBMS relacional	Mecanismo para determinar el "efecto lateral" de un evento.
Estrategia de Optimización para la evaluación de reglas	Procesamiento concurrente de reglas	Evaluación repetida de la restricción violada		Treat / eventos lógicos	Treat / eventos lógicos
Especificación de eventos	Tiene un lenguaje de eventos	Se considera un evento cuando se crea / accede a un objeto.	Son los mensajes que aparecen en la cláusula CONTEXT	Son Oper. sobre las relaciones: update,delete, append, retrieve o Eventos de tiempo	Son generados por los métodos que modif. a los Objetos. Se determina automáticamente en compilación

CAPÍTULO 7
CONCLUSIONES DEL TRABAJO

Este trabajo es el resultado de una investigación originada a partir de la idea de integrar una BDOO con mecanismos de control que incrementen la potencialidad de un DBMS convencional, para que el mismo este preparado, por ejemplo, para soportar un control automático de restricciones.

En el primer capítulo se vieron las características de los sistemas de bases de datos orientados a objetos, en el segundo se revisó la importancia de la imposición automática de consistencia para el desarrollo de aplicaciones de base de datos. La literatura existente sobre el tema coincide en proponer, como la solución del problema, la incorporación de algún sistema de reglas en el DBMS. Esto da origen a un nuevo paradigma de bases de datos: las bases de datos activas.

Después de haber estudiado los puntos más importantes de las bases de datos activas en forma general, sin tener en cuenta un DBMS particular, el objetivo de este trabajo se convirtió en definir las características a incorporar en un OODBMS para transformarlo en activo. A fin de poder analizar con mayor claridad puntos principales de la propuesta se tomó como referencia a SIGMA, un *Modelo Unificado para Bases de Datos Orientadas a Objetos* [SmTa93]. Sin embargo, los fundamentos expresados podrían ser adaptados para otros modelos.

Debido a que SIGMA no cuenta con un prototipo implementado, las ideas expresadas en el presente Trabajo de Grado son también una definición conceptual sin su implementación.

La contribución de esta propuesta es haber integrado coherentemente los distintos elementos para lograr un modelo de base de datos activa, describiendo desde la especificación de restricciones hasta el modelo de ejecución.

Antes de realizar la propuesta de un sistema de base de datos activo, se analizaron las cuestiones más relevantes que debían ser resueltas para comprender las implicancias de la integración de un sistema de reglas dentro de un DBMS. Los aspectos desarrollados en los primeros capítulos surgen de conceptualizar el problema desde los trabajos existentes más representativos sobre el tema.

Un punto muy discutido por los distintos grupos de investigación sobre bases de datos activas orientadas a objetos es el rol que juegan las reglas en el modelo de datos, habiendo quienes sostienen la idea de modelizarlas como objetos. En este trabajo se adopta otra representación, donde las reglas se especifican junto con la definición de los objetos, es decir forman parte de las entidades.

Además, aquí se sugiere una estrategia flexible, adaptada desde otras propuestas, para administrar el chequeo de las restricciones desde los programas, ofreciendo constructores para que el programador tenga cierto control del momento de interacción entre el sistema de transacciones y el intérprete del sistema de reglas.

Están bien establecidas la forma de detectar los eventos de base de datos, así como también la adaptación de la red de discriminación y el algoritmo de TREAT, para implementar el intérprete del sistema de reglas.

Un aporte destacable de la propuesta es el análisis sobre la composición de objetos y su efecto en el control de las restricciones. Se encontró una solución transparente en la cual, ni la definición de objetos, ni la especificación de restricciones se ven afectados.

ANEXO I: INTÉRPRETES DE SISTEMAS DE REGLAS

El modelo de ejecución de un DBMS activo basado en un sistema de reglas debe soportar un *máquina de inferencia* para determinar que reglas se activan ante un determinado evento y ejecutar las acciones especificadas.

El problema de los intérpretes de sistemas de reglas, el costo computacional del algoritmo de macheo que detecta las reglas cuyas condiciones son satisfechas por el estado de la memoria de trabajo, está presente en los DBMS activos.

Recordando que la formulación del aspecto de confrontación es: dado un conjunto de reglas, el corriente estado de la memoria de trabajo, el corriente conjunto de conflicto y un conjunto de cambios a ser introducidos en la memoria de trabajo, cuál es el mejor modo para determinar los cambios en el conjunto de conflicto que resultan a partir de los cambios en la memoria de trabajo?

El costo *C* de un algoritmo de 'matching' [Mira90] esta dado por: $C=k(c+s)$, donde *k,c,s* representan el número de comparaciones por cambio en la memoria de trabajo, costo de una comparación y costo de mantener el estado.

McDermott, Newell y Moore [Mira90] han identificado tres tipos de información de conocimiento que puede ser incorporado en un algoritmo para sistemas de reglas para ganar eficiencia. Ellos son:

Condición de pertenencia: Provee conocimiento acerca de la posible satisfacción de cada elemento de condición individual. El algoritmo puede ignorar aquellas reglas que no están activas, esto es, las que no tienen cada uno de sus elementos de condición positiva parcialmente satisfecho.

Soporte de memoria: Provee conocimiento acerca de cuales elementos de la memoria de trabajo, satisfacen parcialmente cada elemento de condición individual. Es similar a la condición de pertenencia, pero en lugar de mantener un contador de los elementos de la memoria de trabajo macheados, se guarda el subconjunto de dichos elementos para cada patrón de condición. Esta representación es llamada alfa-memorias.

Relación de condición: Provee conocimiento acerca de la interacción de elementos de condición dentro de una regla, y la satisfacción de ellas. Esta representación es llamada beta-memorias. Su efecto es similar a la unión (join) de dos tablas en el modelo relacional.

El algoritmo de Rete es el más comúnmente usado para la implementación de sistemas de reglas. Este incorpora soporte de memoria y relación de condición. El algoritmo de TREAT incorpora una nueva fuente de información, soporte de conjunto de conflicto para mejorar la eficiencia de los sistemas de reglas.

Seguidamente se presenta una descripción de los conceptos básicos del algoritmo de Rete y luego el de TREAT.

I - El algoritmo de Rete [For82]

En un intérprete de sistemas de reglas, la salida del proceso de macheo y la entrada a la resolución de conflicto es un conjunto llamado *conjunto de conflicto*. Este es una colección ordenada de pares de la forma < Regla, Lista de elementos macheados por su LHS>.

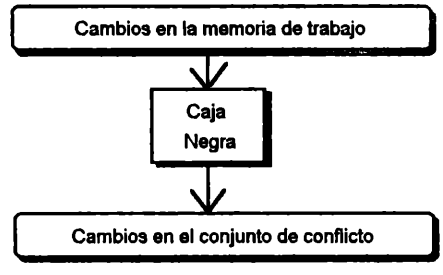
Una regla consiste de un 'antecedente', comúnmente llamado premisa, contenido en la parte **IF**, o parte izquierda (LHS) y un 'consecuente', también llamado conclusión, contenido en la parte **THEN**, o parte derecha (RHS).

Los pares ordenados son llamados instanciaciones. El algoritmo compara un conjunto de LHS contra un conjunto de elementos para descubrir todas las instanciaciones, sin iterar sobre los conjuntos.

I.1 - Como evitar la iteración sobre la memoria de trabajo

El paso que puede requerir iteración es determinar si un patrón dado machea alguno de los elementos de la memoria de trabajo. Los intérpretes más simples determinan esto comparando el patrón contra los elementos, uno a uno. La iteración puede ser evitada almacenando, con cada patrón, una lista de los elementos que machean con él. Las listas son actualizadas cuando la memoria de trabajo cambia. Cuando un elemento entra a la memoria de trabajo, el intérprete encuentra todos los patrones con los que machea y lo agrega a sus listas. Cuando un elemento sale de la memoria de trabajo, el intérprete de nuevo encuentra todos los patrones con los que macheaba y lo elimina de sus listas.

El macheador de patrones puede ser visto como una caja negra con una entrada y una salida:



Máquina de Inferencia

La caja negra recibe información acerca de los cambios que son introducidos en la memoria de trabajo, y determina los cambios que deben ser hechos en el conjunto de conflicto para mantenerlo consistente.

Las descripciones de los cambios en la memoria de trabajo, que son pasados a la caja negra son llamados *tokens*. Un token es un par ordenado de un signo y una lista de elementos de datos. El signo indica si el elemento debe ser agregado o eliminado de la memoria de trabajo.

I.2 - Como evitar la iteración sobre la memoria de reglas

El algoritmo de Rete evita la iteración sobre el conjunto de producciones usando una red de almacenamiento estructurada en árbol, para las producciones. La red,

compilada desde los patrones de las producciones, es el principal componente de la caja negra.

Compilación de patrones: Cuando el intérprete procesa un elemento de la memoria de trabajo, testea varias características del elemento. Estas pueden ser divididas en dos clases:

1 - Las características intra-elemento, son las que involucran sólo un elemento de la memoria de trabajo.

2 - Las características inter-elemento, resultan de tener ocurrencias de una variable en más de un patrón.

El compilador de patrones construye una red ligando los nodos que testean elementos con aquellas características. Cuando el compilador procesa una LHS, determina las características intra-elemento que cada patrón requiere y construye una secuencia lineal de nodos para ese patrón. Después construye los nodos para testear las características inter-elemento. Cada uno de los nodos tienen dos inputs, así que ellos pueden unir dos caminos de la red en uno. Finalmente, después de los nodos de doble input, el compilador construye un nodo terminal para representar la regla.

Procesamiento en la red: El nodo raíz de la red es la entrada a la caja negra. Este recibe los tokens que son enviados a la caja negra y pasa copia de los tokens a todos sus sucesores. Los sucesores del nodo tope de la red, son los nodos para realizar los testeos intra-elemento, tienen un única entrada y una o más salidas. Cada nodo testea una característica y envía los tokens que pasan el testeo a sus sucesores. Los nodos de doble entrada comparan tokens de diferentes caminos y los unen en tokens más grandes si ellos satisfacen las restricciones inter-elemento de la LHS. A causa de los testeos realizados por los otros nodos, un nodo terminal recibirá sólo tokens que instancian la LHS. El nodo terminal envía fuera de la caja negra la información que el conjunto de conflicto debe ser cambiado.

Salvando información en la red: La caja negra debe mantener el estado de la información porque debe conocer que es lo que hay en la memoria de trabajo. En las redes de Rete todo el estado es almacenado en los nodos de doble entrada. Cada uno

de ellos contiene dos listas, una con los elementos de su entrada izquierda, otra con los de la derecha. Los tokens son almacenados mientras ellos son útiles.

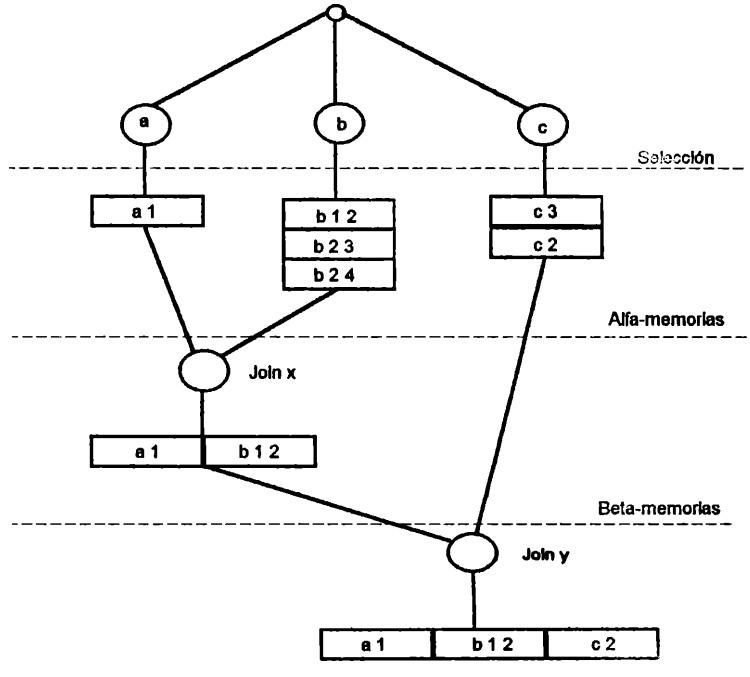
Usando los signos(tags): El signo de un token indica como debe ser cambiado el estado de la información cuando el mismo es procesado. Los tokens + y - son procesados de la misma manera excepto que:

a) El nodo terminal usa el signo para determinar si agregar o remover una instanciación del conjunto de conflicto.

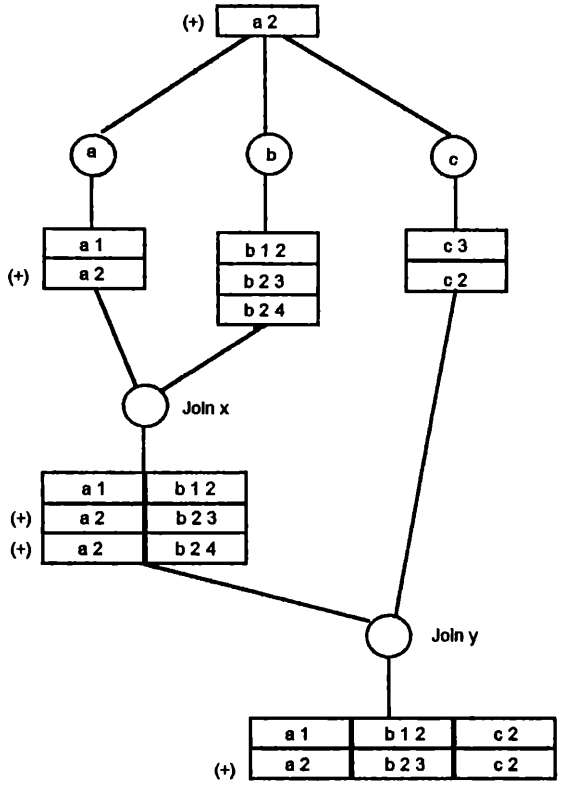
b) Los nodos de doble entrada usan el signo para determinar como modificar su estado de memoria interna. Cuando un token de signo + es procesado, se almacena en la memoria, si es uno de signo - se elimina de ella.

c) Los nodos de doble entrada usan el signo para determinar el signo del token que ellos construyen. Cuando un nuevo token de salida es creado, toma el signo del token que arribó al nodo.

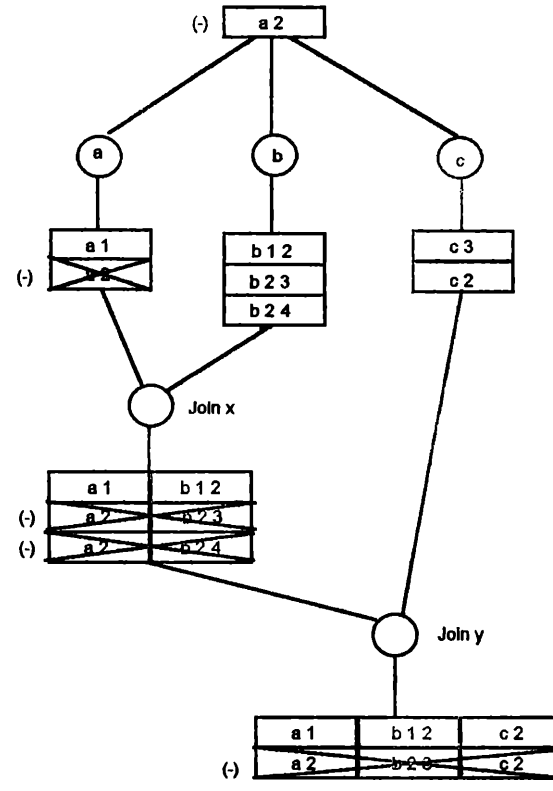
En los siguientes tres gráficos se puede observar la actividad de una red de RETE. La regla considerada es: a(x) & b(x,y) & c(y)



Estado inicial de una red de RETE



Incorporación de un token en una red de RETE



Borrado de un token en una red de RETE

II - El algoritmo de TREAT [Mira90]

El algoritmo de TREAT incorpora una nueva fuente de información, llamada *Soporte de Conjunto de Conflicto*. La observación clave para el desarrollo de TREAT es que diferentes beta-memorias en la red de RETE mantienen el mismo estado. La información presente en una beta-memoria puede aparecer en una beta-memoria más profunda en la red. Además, el conjunto de conflicto contiene mucha de la información almacenada en las beta-memorias.

TREAT mantiene el conjunto de conflicto a lo largo de los distintos ciclos del sistema de reglas, y usa su contenido para reducir el número de comparaciones requeridas para evaluar las reglas.

II.1 - Consideraciones

Para tomar ventaja de mantener el conjunto de conflicto, el algoritmo de TREAT de considera cuatro casos. Para analizarlos se supone el sistema en el siguiente est

Memoria de Reglas	Memoria de Trabajo	Conjunto de Conflicto
a(x) & ¬b(x)	a(1) a(2) b(2)	a(1) & ¬b(1)

1 - Si se agrega un elemento a la memoria de trabajo, que satisface un patrón de condición positivo, el conjunto de conflicto queda igual, excepto por la adición de nuevas instanciaciones de reglas conteniendo los nuevos elementos de la memoria de trabajo.

Memoria de Reglas	Agregar instancias de patrón, positivo en una condición	Memoria de Trabajo antes	Memoria de Trabajo después	Conjunto de Conflicto antes	Conjunto de Conflicto después
a(x) & ¬b(x)	a(3)	a(1) a(2) b(2)	a(1) a(3) a(2) b(2)	a(1) & ¬b(1)	a(1) & ¬b(1) a(3) & ¬b(3)

2 - Si un elemento de la memoria de trabajo, que satisface patrón positivo es eliminado, no se instanciará ninguna nueva regla, y aquellas que contengan el elemento a eliminar serán removidas del conjunto de conflicto

Memoria de Reglas	Eliminar instancias de patrón, positivo en una condición	Memoria de Trabajo antes	Memoria de Trabajo después	Conjunto de Conflicto antes	Conjunto de Conflicto después
a(x) & ¬b(x)	a(1)	a(1) a(3) a(2) b(2)	a(3) a(2) b(2)	a(1) & ¬b(1) a(3) & ¬b(3)	a(3) & ¬b(3)

3 - Si se agrega un elemento a la memoria de trabajo que satisface un patrón de condición negado, puede haber instanciaciones de reglas invalidadas y que deban ser removidas del conjunto de conflicto.

En el caso 2 las instanciaciones invalidadas están explícitamente contenidas en el conjunto de conflicto. En este, el tercer caso, la instanciación invalidada no contendrá el elemento de la memoria de trabajo.

Memoria de Reglas	Agregar instancias de patrón, negado en una condición	Memoria de Trabajo antes	Memoria de Trabajo después	Conjunto de Conflicto antes	Conjunto de Conflicto después
a(x) [illegible] ¼b(x)	b(3)	a(3) a(2) b(2)	a(3) b(3) a(2) b(2)	a(3) & ¬b(3)	

4 - Cuando se borra un elemento de la memoria de trabajo, que satisface un pa[illegible] el cual aparece negado en una condición, puede provocar que entren al conjunto d[illegible]flicto nuevas instanciaciones de reglas.

Memoria de Reglas	Borrar instancias de patrón, negado en una condición	Memoria de Trabajo antes	Memoria de Trabajo después	Conjunto de Conflicto antes	Conjunto de Conflicto después
a(x) & ¡b(x)	b(2)	a(3) b(3) a(2) b(2)	a(3) b(3) a(2)		a(2) & ¼b(2)

II.2 - Algoritmo detallado

TREAT usa condición de pertenencia, soporte de memoria y soporte de conjunto de conflicto. Las alfa-memorias en lugar de existir en una red, están en un vector, donde cada entrada contiene una de ellas.

Las alfa-memorias son divididas en tres particiones: anteriores, nuevos a eliminar y nuevos a agregar, que forman tres vectores separados: la memoria anterior, que contiene los elementos macheados parcialmente que fueron procesados durante los ciclos previos. Durante la fase de acción, los elementos no son agregados a la memoria anterior, sino a las [illegible]emorias en las particiones de nuevos a agregar y de nuevos a eliminar.

Usa una [illegible]ción de hash, cuyo argumento es el primer valor en un elemento de la memoria de trabajo, para ubicar la alfa-memoria que representa la clase a la cual pertenece. En la propuesta hecha en el capítulo 4, esta función es re[illegible]esentada por la capa de la red de discriminación correspondiente al grafo de tipos.

TREAT, además, toma ventaja del soporte de condición. Asociado con cada regla, hay una propiedad llamada 'regla activa'. El algoritmo explícitamente mantiene el conjunto afectado. Cuando una memoria anterior es actualizada, se realiza un testeo para ver si su tamaño se vuelve cero, o se vuelve distinto de cero. Si detecta una vuelta a cero, examina el tamaño de cada una de las memorias anteriores de la regla, y modifica convenientemente la propiedad de regla activa.

Si una alfa-memoria de una regla activa se altera, y el cambio corresponde a uno de los tres casos donde se requiere una búsqueda por instanciaciones, entonces la bú eda tiene ar entre la alfa-memoria modificada (la única con una partición 'r), y las orias anteriores que correspondan a los patrones restantes de la r

) Conjunto CAMBIOS odificaciones a introducir sobre la memoria de trabajo(MT)
) Para cada cambio en el conjunto CAMBIOS (Token) hacer
    Para cada condición que pueda ser afectada por el token corriente hacer
        Si satisface la condición intra-nodo entonces
            Si el token es de signo positivo entonces
                Agregarlo en la lista de ***Nuevos a Agregar*** de ese intra-nodo
            Sino
                Agregarlo en la lista de ***Nuevos a Eliminar*** de ese intra-nodo
            FinSi
        FinSi
    FinPara
FinPara
* Procesar Nuevos a Eliminar*
2) Para cada intra-nodo con su lista de ***Nuevos a Eliminar*** no vacía hacer
    2.1) Borrar de la lista de ***Memoria Anterior*** del intra-nodo corriente los elementos que estén presentes en la lista de ***Nuevos a Eliminar***
    2.3) Si la lista de ***Memoria Anterior*** es vacía entonces
        Actualizar el estado de ***Regla-Activa*** de la regla en la que interviene el intra-nodo corriente
    FinSi
    2.4) Si el intra-nodo es positivo entonces
        Buscar en el ***Conjunto de Conflicto*** instanciaciones de reglas que contengan los elementos a eliminar, y removerlas
    Sino
        Si la regla corriente está activa entonces
            Realizar la búsqueda por nuevas instanciaciones
        FinSi
    FinSi
FinPara
* Proce r Nuevo Agregar *
3) Par a intr do con su lista vos a Agreg no vacía hacer
        Si la de ***Memoria A*** es vacía entonces
            A zar el estado de ***Regla-Activa*** de la regla en la que interviene el intra-nodo
            nte
        Fi i
    3.2) Agregar en la lista de ***Memoria Anterior*** del intra odo corriente los elementos que estén presentes en la lista de ***Nuevos a Agregar***
    3.3) Si la regla está activa entonces
        Realizar la búsqueda por nuevas instanciaciones
    FinSi
    3.4) Si el intra-nodo es positivo entonces
        Agregar aquellas instanciciones de la regla en el ***Conjunto de Conflicto***
    Sino
        Buscar en el ***Conjunto de Conflicto*** instanciaciones de reglas que contengan los elementos a eliminar, y removerlas
    FinSi
FinPara

Algoritmo para evaluar un cambio en la memoria de trabajo: procesamiento de un token en la red

REFERENCIAS

BraMi93 David a Brand and Daniel Miranker
INDEX SUPPORT FOR RULE ACTIVATION
University of Texas at Austin, Usa

CET93 Bogdan Czejdo, Christoph F. Eick and Malcolm Taylor
INTEGRATING SETS, RULES, AND DATA IN AN OBJECT - ORIENTED ENVIRONMENT (TANGUY).
IEEE Expert, February 1993.

CeWi92 Stefano Ceri, Jennifer Windom
DERIVING PRODUCTION RULES FOR CONSTRAIN MAINTENANCE
IBM Almaden Reseach Center, San Jose C.A. 95120

CFPT93 Stefano Ceri, Piero Fraternali, Stefano Paraboschi, Letizia Tanca
AN ARCHITECTURE FOR INTEGRITY CONSTRAIN MAINTENANCE IN ACTIVE DATABASES.
Dipartimento di Elettronica e Informazione
Politecnico de Milano

CFPT94 Stefano Ceri, Piero Fratemali, Stefano Paraboschi, Letizia Tanca
CONSTRAIN ENFORCEMENT THROUGH PRODUCTION RULES PUTTING ACTIVE DATABASES AT WORK
Dipartimento di Elettronica e Informazione
Politecnico de Milano.

Date90 C.J. Date
INTRODUCTION TO DATABASE SYSTEMS
Addison-Wesley Publishing Company

DUHK94 Suzzane W. Dietrich, Susan D. Urban,
John V. Harrison, Anton Karadimce
A DOOD RANCH AT ASU: INTEGRATING ACTIVE, DEDUCTIVE AND OBJECT ORIENTED DATABASES
University of Arizona State

DHL90 Umeshwar Dayal, Meichun Hsu, Rivka Ladin
ORGANIZING LONG - RUNNING ACTIVITIES WITH TRIGGERS AND TRANSACTIONS.

Dur94 Durkin John
EXPERT SYSTEMS DESIGN AND DEVELOPMENT
University of Akron
Macmillan Publishing Company

ECO93 Nina Edelweiss, José M. V. de Castilho, José M. de Oliveira
A TEMPORAL LOGIC LANGUAGE FOR TEMPORAL CONDITIONS DEFINITION.
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Caixa Postal 15064
91501 - Porto Alegre - RS

EiWe93 Christoph F. Eick, Paul Werstein
RULE-BASED CONSISTENCY ENFORCEMENT FOR KNOWLEDGE- BASED SYSTEMS
IEEE Transactions on Knowledge and Data Engineering, Vol. 5, Feb. 93

For82 C.L. Forgy
RETE : A FAST ALGORITHM FOR THE MANY PATTERN / MANY OBJECTS PATTERN MATCH PROBLEM.
Artificial Intelligence, 19:17-37, 1982

GeJa93 Gehani N.H. and Jagadish
ACTIVE DATABASE FACILITIES IN ODE.
AT&T Bell Laboratories, Murray Hill, NJ

Ham81 Hamilton A. G.
LOGICA PARA MATEMATICOS
Editorial Paraninfo, Madrid

Han89 Eric N. Hanson
AN INITIAL REPORT ON THE DESIGN OF ARIEL : A DBMS WITH AN INTEGRATED PRODUCTION RULE SYSTEM.
Sigmod Record, Vol. 18 No. 3, September 1989.

Han92 Eric N. Hanson.
RULE CONDITION TESTING AND ACTION EXECUTION IN ARIEL
Database Systems Research and Development Center Computer and Information Sciences Department, University of Florida, Gainesville,
FL 32611

Kim95 Kim, Won
NEXT-GENERATION DATABASE TECHNOLOGY
Capítulo 1, 13 y 21

LiLe89 T.W. Ling and S.Y. Lee
INTEGRITY CHECKING FOR TRANSACTIONS IN RELATIONAL DATABASES.
Department of Information Systems and Computer Science National University of Singapore, Lower Kent Ridge, Singapore 0511,Singapore.

McCar89 Dennis R. McCarty
THE ARCHITECTURE OF AN ACTIVE DATA BASE MANAGEMENT SYSTEM (HIPAC).
Xerox Advanced Information Technology 4 Cambridge Center, Cambridge MA 02139

Mira90 Miranker, Daniel P.
TREAT : A NEW AND EFFICIENT MATCH ALGORITHM FOR AI PRODUCTIONS SYSTEMS.
Morgan Kaufmann Publishers, Inc., San Mateo California.

SLR88 Timos Sellis, Chin - Chen Lin and Louiqa Raschid
IMPLEMENTING LARGE PRODUCTIONS SYSTEMS IN A DBMS ENVIRONMENT : CONCEPTS AND ALGORITHMS
University of Meryland, Colledge Park, MD 20742.

SmTa93 Smith Clara, Tau Carlos
UN MODELO UNIFICADO PARA BASES DE DATOS ORIENTADAS A OBJETOS
U.N.L.P. 1993

Ston92 Stonebraker M.
THE INTEGRATION OF RULE SYSTEM AND DATABASE SYSTEMS
IEEE Transactions on Knowledge an Data Engineering 4:5 pp. 415,423

Wid93 Jennifer Widom
DEDUCTIVE AND ACTIVE DATABASES : TWO PARADIGMS OR ENDS OF A SPECTRUM ?
IBM Research Division

WiFi90 Jennifer Windom, Sheldon J. Finkelstein
SET ORIENTED PRODUCTION RULES IN RELATIONAL DATABASE SYSTEMS.

BIBLIOGRAFÍA

BuTh90 J.C. Burneau, O.Thiery
OBJECT ORIENTED DATABASES FOR MAINTENANCE EXPERT SYSTEMS.
TéléDiffusion de France Cerlor 1, rue Marconi F-67070 Metz.

ChKa91 Qiming Chen and Yahiko Kambayashi.
NESTED RELATION BASED DATABASE KNOWLEDGE REPRESENTATION.
Computer Science Dept. University of California, Los Angeles, USA.
Computer Science Dept. University of Kyoto, Kyoto, JAPAN .

Coh89 Donald Cohen
COMPILING COMPLEX DATABASE TRANSITION TRIGGERS
Information Sciences Institute University of Southern California.

Don93 Donovan Hsich
A LOGIC TO UNIFY SEMANTIC - NETWORK KNOWLEDGE SYSTEMS WITH OBJECT - ORIENTED DATABASE MODELS.
Computer Science Laboratory, SRI International, Menlo Pack, CA 94025.

GF83 Anoop Gupta and Charles L. Forgy.
MEASUREMENTS ON PRODUCTION SYSTEMS.
Technical Report CMU-CS-83-167, Carnegie-Mellon University, December 1983

GKT91 Güntzer, W. Kiebling, H. Thöne
NEW DIRECTIONS FOR UNCERTAINTY REASONING IN DEDUCTIVE DATABASE (DATALOG)
1991 Acm.

JaJe89 Matthias Jarke, Manfred Jeusfeld
RULE REPRESENTATION AND MANAGEMENT IN CONCEPBASE
Fakultät für Mathematic und Informatik, Universität Passau, Postfach 2540, F.R. Germany

LoOz91 Yanjun Lou and Meral Ozsoyoglu
LLO : AN OBJECT - ORIENTED DEDUCTIVE LANGUAGE WITH METHODS AND METHOD INHERITANCE.
Department of Computer Engineering and Science Case Western Reserve University Cleaveland, OH 44106, Usa.

MedAn92 Claudia Bauser Medeiros, Márcia Jacobina Andrade
IMPLEMENTING INTEGRITY CONTROL IN ACTIVE DATABASES
Relatório Técnico DCC - 06 / 92

MGW92 Claudia Bauzer Medeiros, Geneviève Jomier, Wojciech Cellary
MAINTAINING INTEGRITY CONSTRAINS ACROSS VERSIONS IN A DATABASE.
Relatório Técnico DCC - 08 / 92

ScCa92 Ulrich Schiel and André Felipe de Carvalho
TOM - RULES : A UNIFORM AND FLEXIBLE APPROACH TO EVENTS, CONSTRAINS AND DERIVED INFORMATION.
Federal University of Paraíba, Departamento de Sistemas de Computacão.

SyBase91 SYBASE V4.2 REFERENCE MANUAL
Sybase Corp. Emeryville, CA

VPT91 F. Y. Villemin, A. Paoli, Tourrilhes & M. Le
THE ALDOUS90 PROYECT : MERGING OBJECT - ORIENTED DATABASES AND KNOWLEDGE - BASED SYSTEMS
Centre D'eture et de recherche en informatique du Cnam 292, rue Saint Martin 75141 Paris Cedex03 (France)

Yoon90 Jong P. Yoon
DATABASE UPDATES USING ACTIVE RULES : A UNIFIED APPROACH FOR CONSISTENCY MAINTENANCE.
School of information Technology and Engineering George Mason University, FairFax, VA 22030-4444.