Trabajo fin de grado

Soporte a la decisión clínica en la farmacovigilancia

Directores: Manuel Campos – José Manuel Juárez

Autor: Tony Wang Chen – X1968208Y

2013-2014

FACULTAD DE INFORMÁTICA – UNIVERSIDAD DE MURCIA

RESUMEN

Según la Organización Mundial de la Salud, la farmacovigilancia “trata de recoger, vigilar, investigar y evaluar la información sobre los efectos de los medicamentos, productos biológicos, plantas medicinales y medicinas tradicionales, con el objetivo de identificar información sobre nuevas reacciones adversas y prevenir los daños en los pacientes”. Tanto a nivel nacional como Europeo, existe una legislación que regula el uso de medicamentos, y que establece como aspecto clave la farmacovigilancia. En España, éste es uno de los objetivos principales de la Agencia Española de Medicamentos y Productos Sanitarios, dependientes de Ministerio de Sanidad, Servicios Sociales e Igualdad.

Los actores participantes en los procesos de recogida de información sobre efectos adversos son tanto los ciudadanos como los profesionales clínicos.

En un hospital, el número de fármacos administrados a los pacientes es muy alto, y, por tanto, son necesarios muchos recursos para hacer una revisión eficiente de los tratamientos. Por un lado, el volumen de información que los clínicos deben tener a su disposición es muy alto considerando el gran número de fármacos existentes, y la cantidad de información proporcionada por cada uno de ellos. Por otro lado, esta información tiene que ser además contrastada con los datos particulares de cada paciente antes de aplicar un tratamiento.

Un sistema de soporte a la decisión que asista a los clínicos y farmacéuticos aumentaría la eficiencia de la revisión de tratamientos, y, como consecuencia, la seguridad del paciente.

Así, el objetivo de este Trabajo Fin de Grado es el desarrollo de un sistema de soporte a la decisión clínica en farmacovigilancia.

Para llevarlo a cabo, se han definido varios subojetivos. En primer lugar, se modelará todo el conocimiento sobre los medicamentos, haciendo énfasis en los tipos de alertas existentes (por ejemplo, alergias o interacciones medicamento-medicamento) y sus mecanismos de disparo. Para ello, se recurrirá a las normativas nacionales sobre medicamentos y al catálogo de medicamentos aprobados por la Agencia Española de Medicamentos y Productos Sanitarios, y se estudiarán los estándares de representación.

En segundo lugar se desarrollará el sistema de soporte a la decisión clínica. Se estudiará el uso que se ha hecho de técnicas inteligentes para farmacovigilancia, así como del conocimiento requerido para su diseño. Inicialmente, este proyecto usará una aproximación basada en reglas, que han sido tradicionalmente uno de los sistemas más utilizados, incluso desde el inicio del desarrollo de sistemas basados en conocimiento como es MYCIN de Edward Shortliffe. También, se estudiarán las plataformas disponibles para el desarrollo de soporte a la decisión clínica, prestando interés a las plataformas abiertas que den soporte a reglas.

En último lugar, se realizará una evaluación inicial en un dominio real. Se analizarán, entre otros, las limitaciones de la representación de conocimiento elegida para la información de los medicamentos y para el conocimiento dado por las guías clínicas de tratamiento. Otro parámetro de análisis será la capacidad de razonamiento del sistema, y su eficiencia en el dominio seleccionado.

EXTENDED ABSTRACT

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

[TODO] Aquí iría una pequeña introducción al proyecto.

1.1. MOTIVACIÓN

[TODO] En este apartado los motivos por lo que se van a llevar a cabo el proyecto.

2. OBJETIVOS

El principal objetivo de este trabajo final de grado será crear un sistema de soporte a la decisión clínica para la farmacovigilancia. Para ello se tendrán que cubrir una serie de subobjetivos que se nombran a continuación:

- Se estudiarán los tipos de efectos adversos en alertas de vigilancia microbiológica.
- Se estudiarán alternativas para el diseño y desarrollo de alertas en un sistema de información clínico.
- Se modelarán las alertas que se produzcan en farmacovigilancia.
- Se estudiará la plataforma *OpenCDS*.
- Se implementará un prototipo en *OpenCDS*.

Por otro lado, no formará parte de este proyecto las siguientes cuestiones:

- Evaluación clínica de la herramienta (será un prototipo que no estará validado por ningún experto en el campo).
- Modelado de todos los tipos de efectos adversos.
- Extensión de la plataforma *OpenCDS*.
- Módulo de conexión con el SNS.

2.1. METODOLOGÍA Y HERRAMIENTAS

[TODO] En esta sección se explicarán las herramientas que se podrían haber usado para abordar el proyecto y se hará hincapié en aquella que finalmente se ha decidido usar. Se describirá las limitaciones, los estándares... y se harán comparativas. También se mencionarán sistemas que hayan ya existido que hicieran las mismas funciones así como sus características.

2.1.1. SISTEMAS CON OBJETIVOS SIMILARES

[TODO] Tras un una búsqueda y un pre-estudio antes de abordar el problema, se han encontrado una serie de sistemas que tienen un objetivo similar. Los más destacados son los siguientes.

2.1.2. HERRAMIENTAS

[TODO] Estudio de las herramientas y comparativas (tabla). Las herramientas que se estudiarán serán PL/SQL, Java, Reglas (Drools, Jess, Clips), y OpenCDS. Las dos primeras las descartamos directamente indicando las razones.

2.1.3. DESARROLLO UTILIZANDO REGLAS

[TODO] Una breve introducción a por qué se podría realizar utilizando reglas.Explicar JESS, CLIPS, y al final explicar DROOLS. Además introducir que vamos a hablar del algoritmo RETE.

2.1.3.1. ALGORITMO *RETE*

Creado por *Charles Forgy*, es un algoritmo diseñado para SBRs. El nombre proviene de **REdundancia TEmporal** y utiliza una red de nodos denominada **red RETE**. El funcionamiento básico de *RETE* es el siguiente:

1. Realizar los cambios en la base de hechos.
2. Propagar los cambios por la red *RETE*.
3. Modificar el conjunto conflicto.

Surge porque en un SBR en cada ciclo equiparamos todas las reglas con todos los elementos de la base de hechos para formar el conjunto de conflicto y el SBR sería poco eficiente. *RETE* lo que introduce es un proceso de filtrados de reglas antes de generar el conjunto conflicto. Antes de explicar en más detalle *RETE*, se deben saber los aspectos más importantes de un SBR.

Un SBR se inspira en los sistemas de deducción en lógica proposicional o de primer orden. Se entienden las reglas como una sentencia del tipo:

```
IF condición THEN acción
```

Sus componentes son los siguientes:

- **Base de conocimiento**: Contiene las reglas que codifica todo el conocimiento (por norma general es estático y es el conocimiento que se ha ido adquiriendo a lo largo del tiempo). Una regla se disparará cuando se cumplan sus condiciones (antecedentes) y se ejecutará entonces una acción (consecuente).
- **Base de hechos**: Contiene propiamente los hechos ciertos a la hora de resolver el problema. Algunos de estos vendrán como datos de entrada y otros serán conclusiones inferidas (aplicándose las reglas de la base de conocimiento). Esta base es dinámica puesto que se irán añadiendo y eliminando hechos cuando se disparen las reglas.
- **Motor de inferencia**: Se encarga de seleccionar las reglas que se pueden disparar en un momento y las aplica siguiendo unas determinadas condiciones para obtener nueva información y actualizar las bases.
- **Conjunto conflicto**: Conjunto de reglas que pueden ser disparadas en un momento, con la base de hechos que haya.

Por lo general, el algoritmo de un SBR puede funcionar de dos maneras:

- **Encadenamiento hacia delante**: Se busca un conjunto de metas que se verifican a partir de un conjunto de hechos.
- **Encadenamiento hacia detrás**: Se determinan si se va cumpliendo la meta con el conjunto de hechos.

Los pasos que sigue un SBR, por norma general se pueden resumir en los siguientes:

1. Con una base de conocimiento y una base de de hechos inicial, y mientras no se tenga la meta en la base de hechos y se sigan teniendo reglas en el conjunto conflicto hacer:
    a. Meter en el conjunto conflicto aquellas reglas que pueden ser disparadas.
    b. Resolver el conflicto y seleccionar una regla.
    c. Aplicar la regla, sacarla del conjunto conflicto y actualizar los datos con el consecuente de la regla.
    d. Comprobar si está la solución en la base de hechos. Si es así, terminar.

Una vez explicado esto, se sabe que se puede situar el algoritmo *RETE* en el primer paso del algoritmo general de un SBR. Generar un buen conjunto conflicto hará que el SBR sea mucho más eficiente. Como se ha empezado diciendo, *RETE* tiene un filtro que permite que en cada ciclo no sea necesario equiparar todas las reglas con todos los elementos de la base de hechos y

como normalmente los cambios que implica la ejecución de una regla son pocos, hace que el filtro sea eficiente. La comunicación a la red *RETE* de los cambios producidos en la base de hechos se hacen mediante la utilización de **testigos**, los cuales están formados por dos componentes: un signo (+ o -) para indicar si el elemento se ha borrado o añadido a la base de hechos y los elementos que son añadidos o borrados. Por ejemplo:

`-(A C)` que indica que se han eliminado los hechos `A` y `C` de la base.

La red *RETE* se genera al principio de la ejecución del *SBR* y sigue los siguientes pasos:

1. Se consideran los elementos que son constantes y los que, siendo variables, no aparecen en dos o más elementos de condición distintos en la misma regla.
2. Se consideran el resto de los elementos.

Un ejemplo de red *RETE* con la siguiente regla:

`R1: (A B) ∧ (C ?x D ?x)` → …

Sería lo siguiente:

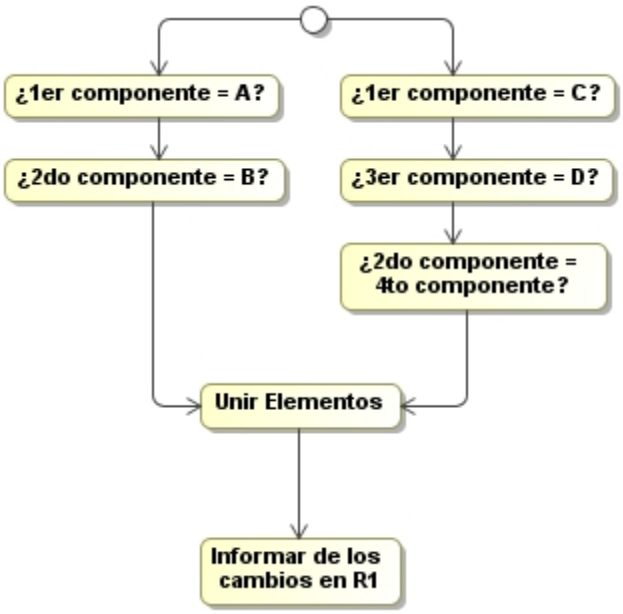

FIGURA 1: EJEMPLO DE RED *RETE*

Se ve que no se hace referencia a las variables ya que se pueden equiparar con cualquier valor que se encuentre en su posición. En este particular caso sólo se ha de comprobar que el segundo componente y el cuarto componente de la segunda parte de la regla coincidan. En caso de que alguna entrada no atraviese la red completa hasta un nodo terminal, quedaría registrada su posición por si en el siguiente ciclo cambiara. La red *RETE* se puede considerar como una función que traslada los cambios de la base de hechos a cambios en el conjunto conflicto. Además se pueden encontrar los siguientes tipos de nodo:

- **Nodo de una entrada** (nodo raíz, nodos de componentes variables o constantes, nodos terminales y nodos de comprobación de variables).
- **Nodos de dos entradas** (se sitúan siempre después de los nodos de una entrada, a excepción de los nodos terminales y pueden ser nodos de unión de elementos condición,

nodos de unión de elementos condición con variables iguales de una misma regla y nodos de unión de los elementos condición negados con los restantes elementos condición de una regla)

En última instancia se tendrá un árbol con todas las reglas de nuestro sistema en forma de red *RETE* y cada vez que se introduzcan o eliminen nuevos hechos se modificará el conjunto conflicto (cuando todos los hechos llegan a algún nodo terminal, que son las reglas). No se va a entrar más en detalle, así que a continuación y para terminar se detallan las características más importantes:

- Como en cada ciclo los cambios en la base de hechos suelen ser pocos aunque los contenidos sean muchos, es muy ineficiente equiparar en cada ciclo toda la base de hechos. En una red *RETE* se almacenan las equiparaciones que se producen en cada ciclo con lo que estas pueden ser reutilizadas en el ciclo siguiente y de esta forma el coste computacional depende de la velocidad de cambio de la base de hechos (que suele ser baja) y no se su tamaño (que suele ser grande.
- Aprovecha la **similitud estructural**. Las condiciones de las reglas tienen muchos patrones similares aunque no idénticos. La red *RETE* agrupa los patrones comunes con el fin de evitar repetir las equiparaciones idénticas y elimina las operaciones redundantes construyendo un único nodo por cada operación distinta y uniendo dicho nodo con aquellos que necesitan dicho resultado.

2.1.3.2. *DROOLS*

Una vez se sabe que la plataforma *OpenCDS* funciona por debajo con *DROOLS* como sistema de inferencia de reglas, se va a hacer un estudio del mismo para ver la eficiencia, escalabilidad, etc… Este estudio se realizará con las versiones 5 y 6, puesto que de una a otra cambian bastantes aspectos importantes como por ejemplo, el algoritmo usado en el motor de inferencias. Además, se tiene que tener en cuenta que la plataforma *OpenCDS* usa la versión 5 de *DROOLS* y se espera que para el futuro se adapte a la versión 6, que en teoría debería de proporcionar mejoras al mismo.

2.1.3.2.1. ALGORITMO *RETEOO* VS ALGORITMO *PHREAK*

Como se ha mencionado anteriormente, el aspecto que más diferencia a ambas versiones es su algoritmo del motor de inferencias. Mientras que *DROOLS* 5 usa un algoritmo *ReteOO*, la nueva versión tiene un algoritmo llamado *PHREAK*. Antes de meternos a la explicación del algoritmo *PHREAK*, vamos a recordar cómo funcionaba el motor de inferencias de las versiones anteriores a la 6 de *DROOLS*. En concreto, nos referimos en este caso al anteriormente mencionado algoritmo *ReteOO* que usa como base el conocido *RETE* pero aplicando una serie de mejoras que se documentan a continuación:

- ***Node Sharing:*** Los nodos compartidos se aplican tanto para el *alpha network* como el *beta network*[1]. En el caso del *beta network* es siempre compartido desde la raíz.
- ***Alpha indexing*:** Los nodos *alpha* que tienen muchos nodos hijo utilizan un mecanismo de búsqueda *hash* para evitar tener que probar cada uno de los resultados.

[1] Charles Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence, 19, pp 17–37, 1982.

- ***Beta indexing:*** Los nodos *join*, *not* y *exist* indexan su memoria usando un *hash*. Esto nos reduce los intentos de unión para las comprobaciones de igualdades. Recientemente se ha añadido la indexación de rango a los nodos *not* y *exists*.
- ***Tree based graphs:*** Las coincidencias de los *join* no contienen referencias a las de su nodo padre o nodos hijos. Por tanto cuando eliminamos tendremos que volver a calcular todas las coincidencias de los *join* de nuevo, lo que implica recrear todos los objetos coincidentes del *join* para que seamos capaces de encontrar las partes de la red en la que debemos de eliminar las tuplas. Todo este método explicado es conocido como *symmetrical propagation*. Un grafo basado en árbol nos proporciona referencias al nodo padre y a los nodos hijos por lo que cuando eliminamos lo único que deberemos de hacer es seguir las referencias que tenemos. De esta manera, tenemos lo conocido como *asymmetrical propagation*. El resultado es que de esta manera lo haremos más rápido y el impacto al recolector de basura será menor, además de ser más robusto porque los cambios en los valores no causan pérdidas de memoria cuando ocurren sin que el motor de inferencias sea notificado.
- ***Modify-in-place:*** El algoritmo *RETE* tradicional implementa el modificar como un borrar e insertar de nuevo el objeto. Esto causa que el recolector de basura limpie todas las tuplas *join* y por tanto muchos se recrean de nuevo como parte de la inserción. Con *modify-in-place* modificaríamos directamente por lo que evitaríamos este problema.
- ***Property reactive:*** También llamado *"new trigger condition"* permite mas reactividad en las actualizaciones de grano fino. Un patrón puede reaccionar a cambios de propiedades específicas e ignorar otras. Esto evita problemas de recursividad y mejora el rendimiento.
- ***Sub-networks:*** Los nodos *not*, *exists* y *accumulate* pueden tener cada uno de ellos anidados elementos condicionales en forma de *sub-networks*.
- ***Backward Chaining:*** Puede funcionar con árboles de derivación de encadenamiento hacia atrás al igual que *Prolog*. La aplicación usa una pila, por lo que no tiene problemas de recursión para grafos grandes.
- ***Lazy Truth Maintenance***
- ***Heap based agenda:*** La agenda usa una cola *binary heap*[2] para ordenar las reglas que se dispararían por relevancia.
- ***Dynamic Rules:*** Las reglas pueden añadirse y eliminarse en tiempo de ejecución mientras que el motor siga rellenándose con los datos.

Una vez se ha explicadas las características de *DROOLS* 5 podemos pasar a definir la última versión que, como bien se ha mencionado antes, cambia el algoritmo para introducir *PHREAK*. Este algoritmo trata de abordar algunos de los principales problemas de *RETE*. Por tanto, no es ningún algoritmo que se haya escrito desde cero, sino que incorpora todo lo existente en *ReteOO* así como las mejoras que tuviera. Aun siendo una evolución del algoritmo *RETE* este no se clasifica ya como una aplicación de la misma, sino que se introduce una nueva clasificación puesto que los cambios que lleva así lo merece.

Los algoritmos *RETE* tienen como características principales (y en donde se buscan las optimizaciones) que son algoritmos *greedy*[3] (produce todas las coincidencias parciales en las

[2] Chris L. Kuszmaul. "binary heap". Dictionary of Algorithms and Data Structures, Paul E. Black, ed., U.S. National Institute of Standards and Technology. 16 November 2009.
[3] Introduction to Algorithms (Cormen, Leiserson, and Rivest) 1990, Capítulo 17 "Greedy Algorithms" p. 329.

reglas) y que está orientado a los datos, por lo que prácticamente todo el trabajo se realiza durante la inserción, la actualización y el borrado de los mismos. Como *RETE* es un algoritmo *greedy*, esto puede ocasionar que en grandes sistemas se produzcan fallos y se desperdicie mucho trabajo (trabajo sobre todo en esfuerzos para encontrar reglas que se puedan disparar y que al final no son disparadas).

Todo lo contrario ocurre con el algoritmo *PHREAK* que se caracteriza por ser del tipo perezoso (las coincidencias parciales se retrasan) y orientado a mejorar el algoritmo. *PHREAK* está inspirado por otra serie de algoritmos (que no limitados por) como por ejemplo *LEAP*, *RETE/UL* y *Collection-Oriented Match*. *PHREAK* además, tiene todas las mejoras que se han comentado anteriormente de *ReteOO*, añadiendo a ellas el siguiente conjunto de mejoras que se detallan a continuación:

- Tres capas de memoria contextual. Una para los nodos, otra para los segmentos y otra para las reglas.
- Reglas, segmentos y nodos basados en vinculaciones.
- Evaluación perezosa de las reglas.
- Evaluación aislada de las reglas.
- Orientado a establecer propagaciones.
- Evaluaciones usando una pila, con opción de pausar y reanudar.

Cuando el motor en *PHREAK* arranca, todas las reglas empiezan estando desvinculadas por lo que no puede ocurrir ninguna evaluación de las mismas. Las operaciones de inserción, actualizado y borrado son encoladas antes de entrar al *beta network*. Se usa una heurística simple para seleccionar la siguiente regla para evaluar, que está basada en la regla que más probabilidad tiene de ser disparada; esto retrasa la evaluación y el disparado de las otras reglas. Las reglas sólo podrán ser vinculadas una vez y será cuando se cumplan todos sus antecedentes, a pesar de que todavía el sistema no haya empezado a trabajar. En su lugar, se crea un objetivo que representa a la regla y se coloca en una cola de prioridad que es ordenada por relevancia. Cada una de estas colas se asocian con una *AgendaGroup* y sólo la *AgendaGroup* activa inspeccionará su cola, sacando de ella aquella el objetivo de la regla con mayor prioridad y sometiéndolo para su evaluación. Por lo tanto, el trabajo realizado cambia desde la fase de inserción, actualizado y borrado hasta la fase *fireAllRules*. Sólo la regla para la cual el objetivo fue creado es evaluada, mientras que las demás reglas potenciales que llevan los mismos hechos se retrasan. Mientras se están evaluando las reglas individualmente *node sharing* se sigue consiguiendo a través del proceso de segmentación que se explica más adelante.

Cada vez que en *RETE* se hace un *join* se crea una tupla (o coincidencia parcial) que se propaga a los nodos hijo. Por esta razón es caracterizado por ser un algoritmo orientado a tuplas. Por cada nodo hijo al que la tupla llegue, intentará llegar al otro lado del nodo y así sucesivamente hasta el final (tal y como se ha explicado en la red *RETE*). Esto lo que genera es un "efecto" de recursión descendente, hiperpaginando la red ya que desde el punto de entrada hasta el *beta network* intenta propagarse hacia debajo, arriba, derecha e izquierda hacia todos los nodos hoja.

Sin embargo la propagación de *PHREAK* es orientado a colecciones en vez de a tuplas. Para cada regla que se está evaluando, se visita el primer nodo y se procesan las operaciones de inserción, borrado y actualizado que estuvieran en la cola. Los resultados se añaden a una colección y se propagan al nodo hijo. En este nodo hijo se vuelven a procesar las operaciones de inserción, borrado y actualizado de la cola y los resultados los introducen en la misma colección

para ser propagado y así sucesivamente hasta que se llegue a un nodo terminal. Con eso se consigue se sólo se haga una pasada (efecto *pipeline*) y se aísla a la regla que se está evaluando. Se consigue un efecto de proceso por lotes que proporciona ventajas de rendimiento en algunas de las reglas (al igual que las sub-redes con las acumulaciones). También de cara al futuro esto permitirá sacarle partido a las máquinas multi-núcleo de varias maneras.

Las "conexiones" y "desconexiones" usan una máscara de bits por capas, basadas en la segmentación de la red. Cuando se está construyendo la red para una regla los segmentos son creados para los nodos que comparten el mismo conjunto de reglas. Una regla está formada por un *path* de segmentos aunque si no es una regla compartida será un elemento único. Una máscara de bit offset es asignada a cada nodo del segmento y otra a cada segmento en el *path* de la regla. Cuando al menos hay una entrada a la hora de la propagación de los datos, el bit del nodo se pone en activo. Cuando cada nodo del segmento tiene su bit activo entonces el bit del segmento se pone también en activo, de la misma manera que si algún bit de algún nodo se desactiva, este también se desactivará. Si cada segmento del camino que sigue la regla tiene su bit activo, entonces se puede decir que la regla se ha "conectado" y se crea el objetivo que se trata de programar la evaluación de dicha regla. La misma técnica es usada para detectar nodos, segmentos y reglas sucios. Esto permite que una regla que ya fue "conectada" que vuelva a ser programada para la evaluación si se considera sucia desde la última evaluación. Con la anterior técnica se asegura que ninguna regla se evalúe cuando hay coincidencias parciales en el antecedente, mientras que en *RETE* clásico esto se produce incluso si el último *join* que se realiza es un resultado vacío. Además, como la evaluación de las reglas se realiza siempre desde el nodo raíz, el bit sucio comentado anteriormente permite que los nodos y segmentos que no estén sucios puedan ser saltados.

RETE tiene una única unidad de memoria, la memoria del nodo. En *PHREAK* existen 3 niveles de memoria lo que permite mucha más comprensión contextual durante la evaluación de una regla.

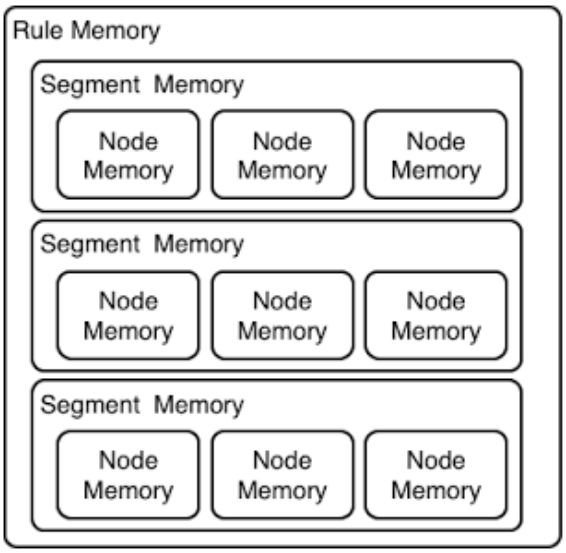

FIGURA 2: CAPAS DE MEMORIA EN *PHREAK*[1]

Todas las evaluaciones de las reglas son incrementales y no desperdician el trabajo recalculando coincidencias que ya se han produciendo. El algoritmo de evaluación es basado en pila en lugar

del método recursivo, por lo que puede ser parado y reanudado en cualquier momento usando un *StackEntry* para representar al nodo actual que se está evaluando. Cada vez que la evaluación de una regla llegue a una sub-red se crea un *StackEntry* para el segmento del *path* exterior y el segmento de la sub-red. El segmento de la sub-red es evaluado primero y cuando la colección (la que se explicó antes que se iba transfiriendo entre nodos) alcance el final del *path* de la su-red se une a una lista de clasificación para que se alimente el nodo exterior. Dicho *StackEntry* entonces se reanuda y puede procesar los resultados de la sub-red. Esto además tiene un beneficio añadido que es que todo el trabajo se procesa en lotes antes de propagarlo al nodo hijo, lo que es mucho más eficiente para los nodos acumulativos.

Este mismo sistema basado en pila se puede usar para un encadenamiento hacia detrás eficiente. Cuando la evaluación de una regla alcanza un nodo *query* se para la evaluación actual, poniéndolo en la pila. Entonces se evalúa la *query* lo que produce un resultado que se guarda en memoria para que al reanudar el *StackEntry* pueda obtenerlo y propagarlo al nodo hijo. Si esta misma *query* llama a otras *queries* el proceso se repetiría pausando la *query* actual y empezando una nueva evaluación de la *query* actual.

Como último punto en el rendimiento, una única regla por norma general no se evaluará más rápido con *PHREAK* que con *RETE*. Para una única regla y los mismos datos ambos crearán las mismas coincidencias y producirán el mismo número de instancias de reglas por lo que tardarán lo mismo. *PHREAK* también se puede considerar más indulgente que *RETE* para las bases de conocimiento que tengan reglas escritas muy pobremente. Cuando la complejidad de las reglas se incrementa, se degrada también el rendimiento. *RETE* produce coincidencias parciales para reglas que no tienen datos en todos sus *joins* y *PHREAK* evita esto. Por tanto no es tanto que *PHREAK* sea más rápido que *RETE* sino que no se ralentiza tanto a medida que el sistema crece.

2.1.4. PLATAFORMA *OPENCDS*

[TODO] Al final se ha utilizado esta herramienta porque…

Las limitaciones que tienen…

El lenguaje de programación y las herramientas de desarrollo…

En este proyecto es importante motivarlo por lo novedoso de la plataforma.

Descripción general de la estructura de OpenCDS a la vez que voy entendiéndolo yo.

3. DISEÑO Y RESOLUCIÓN

Esta sección se corresponde a toda la parte técnica y de programación de la aplicación. Se definirá la aplicación implementada desde su inicio hasta su fin así como

3.1. LA APLICACIÓN USANDO REGLAS

[TODO] Introducción

3.1.1. ESTUDIO DEL RENDIMIENTO CON *DROOLS*: COMPARATIVA DE VERSIONES

Para la siguiente sección, se va a hacer una serie de comparativas de los algoritmos explicados en el apartado 2, en la parte correspondiente a reglas, usando un número determinado de reglas y hechos y midiendo el uso de memoria y CPU así como el tiempo tardado[4] hasta que infieren una solución. Realmente se está comparando íntegramente las versiones de *DROOLS* donde cada una de ellas usa un algoritmo diferente, ya que no se puede aislar y comparar únicamente la parte en la que se hace uso del algoritmo.

Estas pruebas se han realizado bajo un ordenador de las siguientes características:

- **Procesador:** Intel® Core™ i7-3610QM CPU @ 2.30Ghz
- **Memoria RAM:** 8,00 GB
- **Sistema Operativo:** Windows 7 Professional N (SP1) de 64 bits

En primera instancia, se explica aquí los detalles de las pruebas realizadas. Se ha implementado un hecho que contiene un único atributo numérico con valor inicial de 0. Por otro lado, se ha desarrollado un software que genera reglas para nuestra aplicación de pruebas que siguen la siguiente estructura:

```
Rules.drl

      Regla 1: SI hecho.numero=0 ENTONCES hecho.numero=1

      Regla 2: SI hecho.numero=1 ENTONCES hecho.numero=2

      Regla 3: SI hecho.numero=2 ENTONCES hecho.numero=3

      Regla 4: SI hecho.numero=3 ENTONCES hecho.numero=4

      …
```

Dependiendo del número de reglas que se quiera generar y empezando con una regla, cada regla siguiente lo único que hará será modificar el valor del atributo numérico para incrementarlo en una unidad. De esta manera, se consigue que por cada hecho nuevo que se cree, se disparen todas las reglas de la base de conocimiento (a partir de ahora BC). Por tanto, si se tienen dos hechos en la base de hechos (a partir de ahora BH) y tres reglas en la BC, se dispararán un total

[4] Para la medición de la CPU se usa el monitor de sistema de Windows. Para le medición de la RAM se usa el método totalMemory() de Java y por último para la medición del tiempo, se coloca una bandera antes de introducir los hechos en la BH y se comprueba al final de la ejecución cuánto tardó.

de 6 reglas y la aplicación finalizará. Dicho esto, se hacen tablas comparativas según el algoritmo utilizado para las pruebas:

1 Regla / 1 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	46.74/339.12	0.007
***DROOLS* 6**	0.00	42.16/121.5	0.034

1 Regla / 10 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	46.91/339.06	0.008
***DROOLS* 6**	0.00	42.87/121.5	0.039

1 Regla / 100 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	46.72/339.12	0.015
***DROOLS* 6**	0.00	42.85/121.5	0.048

1 Regla / 1000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	47.90/339.00	0.085
***DROOLS* 6**	0.00	44.45/121.5	0.363

1 Regla / 10000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	55.34/339.00	0.636
***DROOLS* 6**	0.00	26.60/121.5	0.675

1 Regla / 50000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	81.45/339.00	1.051
***DROOLS* 6**	0.00	79.84/154.00	0.919

1 Regla / 100000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	121.71/339.12	1.185
***DROOLS* 6**	0.00	81.29/203.00	1.183

1 Regla / 500000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	14.08	352.95/518.56	2.253
***DROOLS* 6**	21.54	270.65/419.5	2.323

1 Regla / 1000000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	17.19	485.38/1041.68	5.7
***DROOLS* 6**	27.09	437.59/691.5	4.058

Con estos primeros resultados se puede ver que como los tiempos de cómputo son tan pequeños no se puede medir bien el uso de CPU que lleva cada ejecución por lo que podría provocar valores incoherentes (los dejamos a 0.00 cuando el tiempo es inferior a 1 segundo).

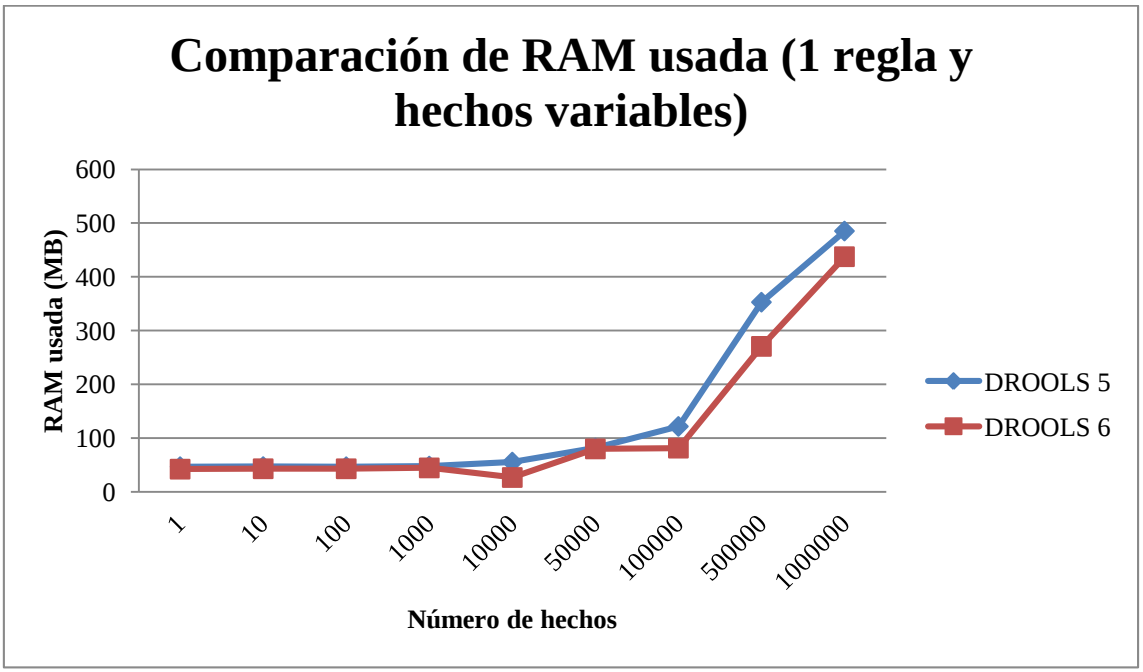

FIGURA 3: COMPARATIVA DE MEMORIA USADA CON 1 REGLA Y HECHOS VARIABLES

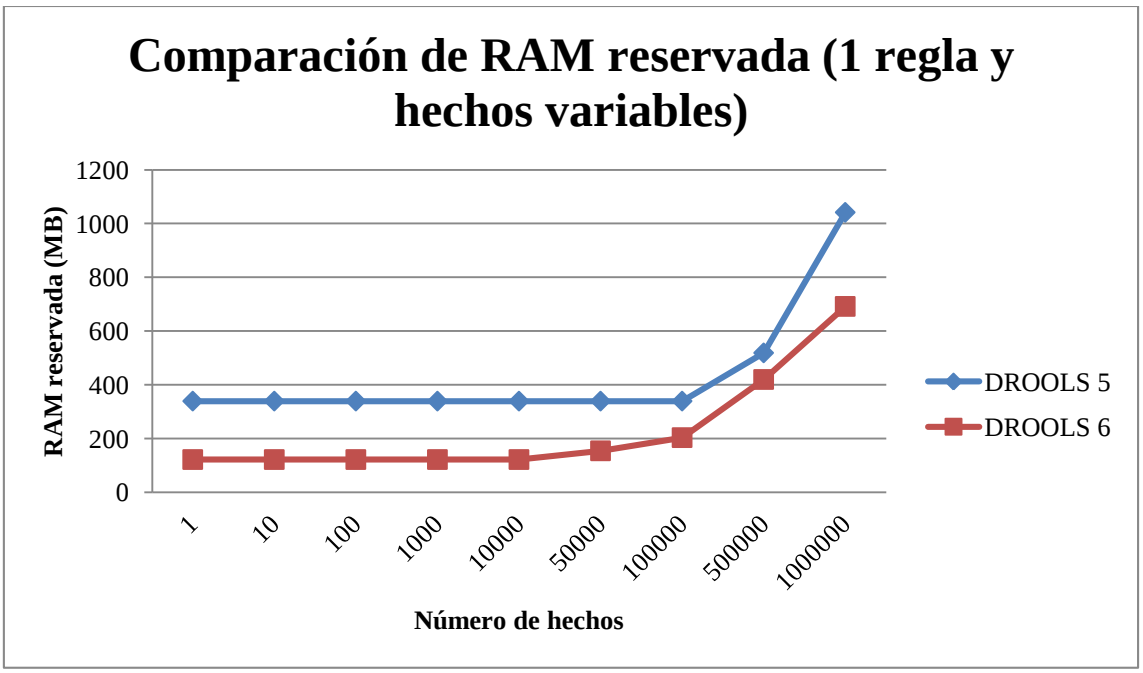

FIGURA 4: COMPARATIVA DE MEMORIA RESERVADA CON 1 REGLA Y HECHOS VARIABLES

Se puede ver que por mucho que aumentemos el número de hechos al tener una regla nada más, la memoria que usa la aplicación no crece tanto (los hechos no necesitan tanta memoria) y el tiempo de cómputo hasta inferir la solución no es tan alto.

La cantidad de memoria que se necesita en *DROOLS* 6 es menor que para la versión 5. Cuando se tienen pocos hechos, parece que *DROOLS* 5 tarda menos en terminar, pero la diferencia es ínfima. A medida que aumentan los hechos vemos que la eficiencia de *DROOLS* 6 aumenta bastante respecto a su antecesor. En cuanto a la CPU, *DROOLS 6* necesita más cómputo y debe

de ser debido a la optimización en el algoritmo que usa. En *DROOLS* 6 también se ve que llegado a un punto, como la memoria está llena el recolector de basura se encarga de vaciarla.

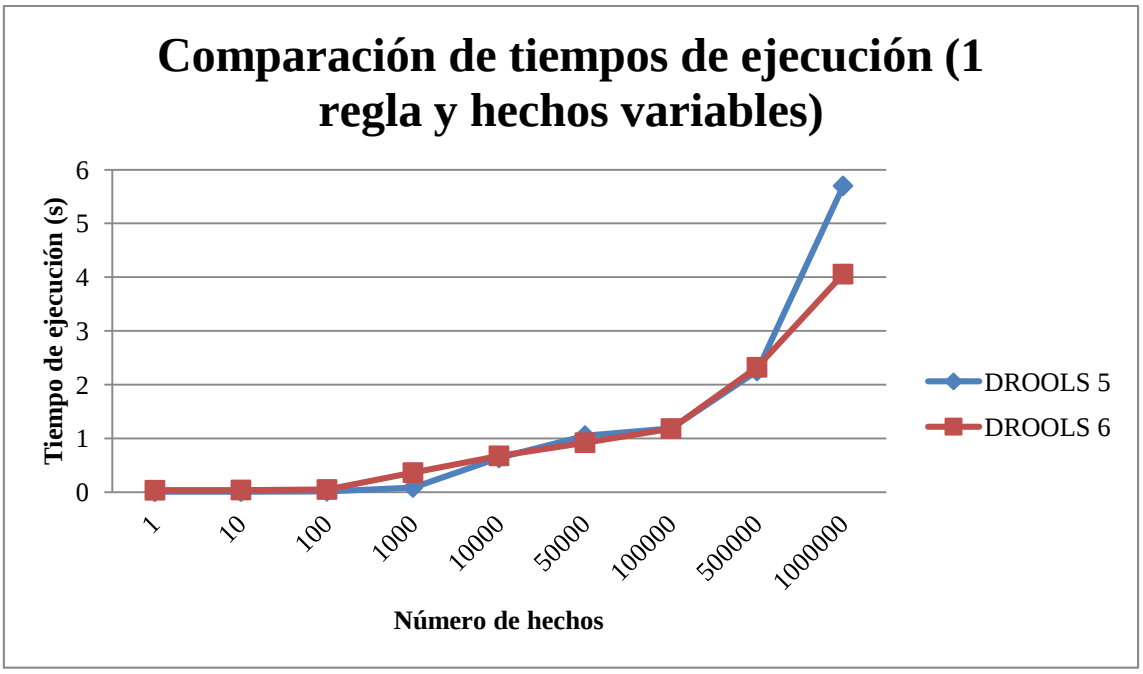

FIGURA 5: COMPARATIVA DE TIEMPOS DE EJECUCIÓN CON 1 REGLA Y HECHOS VARIABLES

En cuanto a tiempos de ejecución no se puede apreciar mucho la diferencia puesto que ambos, en estos ejemplos finalizan muy rápido. No obstante parece ser que a partir de tener bastantes más hechos *DROOLS* 6 empieza a funcionar de manera más rápida.

Se realizan las mismas pruebas pero en vez de aumentar los hechos, esta vez aumentamos las reglas y obtenemos los siguientes resultados.

2 Reglas / 1 Hecho	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	47.89/339.18	0.008
***DROOLS* 6**	0.00	42.34/121.5	0.034

5 Reglas / 1 Hecho	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	48.71/339.12	0.007
***DROOLS* 6**	0.00	43.20/121.5	0.03

10 Reglas / 1 Hecho	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	50.77/339.37	0.008
***DROOLS* 6**	0.00	44.34/121.5	0.034

50 Reglas / 1 Hecho	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	62.33/339.75	0.014
***DROOLS* 6**	0.00	34.05/121.5	0.065

100 Reglas / 1 Hecho	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	79.00/340.25	0.021
***DROOLS* 6**	0.00	80.52/154.0	0.091

200 Reglas / 1 Hecho	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	111.73/341.18	0.037
***DROOLS* 6**	0.00	48.92/154	0.155

500 Reglas / 1 Hecho	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	207.01/343.81	0.08
***DROOLS* 6**	0.00	68.88/152.00	0.305

1000 Reglas / 1 Hecho	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	145.34/347.06	0.153
***DROOLS* 6**	0.00	150.96/225.50	0.522

2000 Reglas / 1 Hecho	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	243.28/497.43	0.305
***DROOLS* 6**	0.00	208.39/353.50	0.945

5000 Reglas / 1 Hecho	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	80.80/787.87	0.913
***DROOLS* 6**	0.00	Out of Memory Error	∞

5500 Reglas / 1 Hecho	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	Out of Memory Error	∞
***DROOLS* 6**	0.00	Out of Memory Error	∞

Se puede ver que el número de reglas que permite la aplicación es más limitada. En estas ejecuciones, el tiempo que tarda en cargar todas las reglas en la BC es bastante grande (el tiempo que sale en la tabla es el tiempo de ejecución). En los datos se observa que las reglas ocupan mucho más en memoria aunque en un punto el recolector de basura trabaje y limpie la memoria. Igualmente en una aplicación real tener más de 2000 reglas no es algo que pueda ser habitual.

Entre *DROOLS* 5 y 6, se cumple también lo mismo que para los hechos. La última versión está más optimizada en cuanto a memoria pero en tiempo tarda más debido a la optimización que sufre. Aun así, las diferencias de tiempo siguen siendo despreciables.

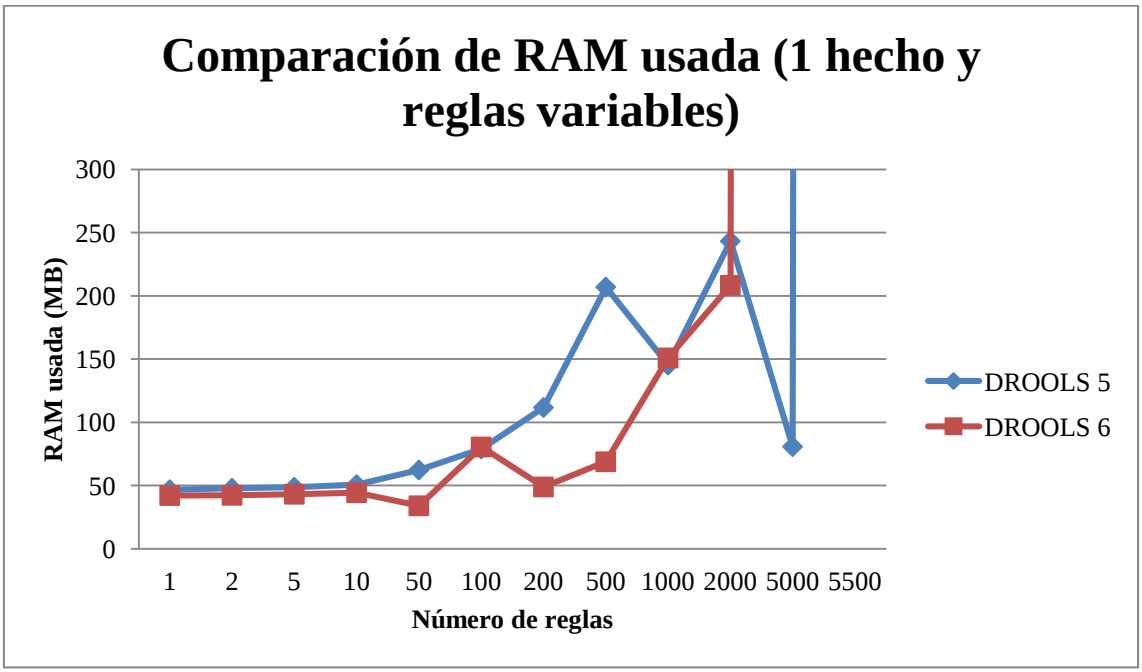

FIGURA 6: COMPARATIVA DE MEMORIA USADA CON 1 HECHO Y REGLAS VARIABLES

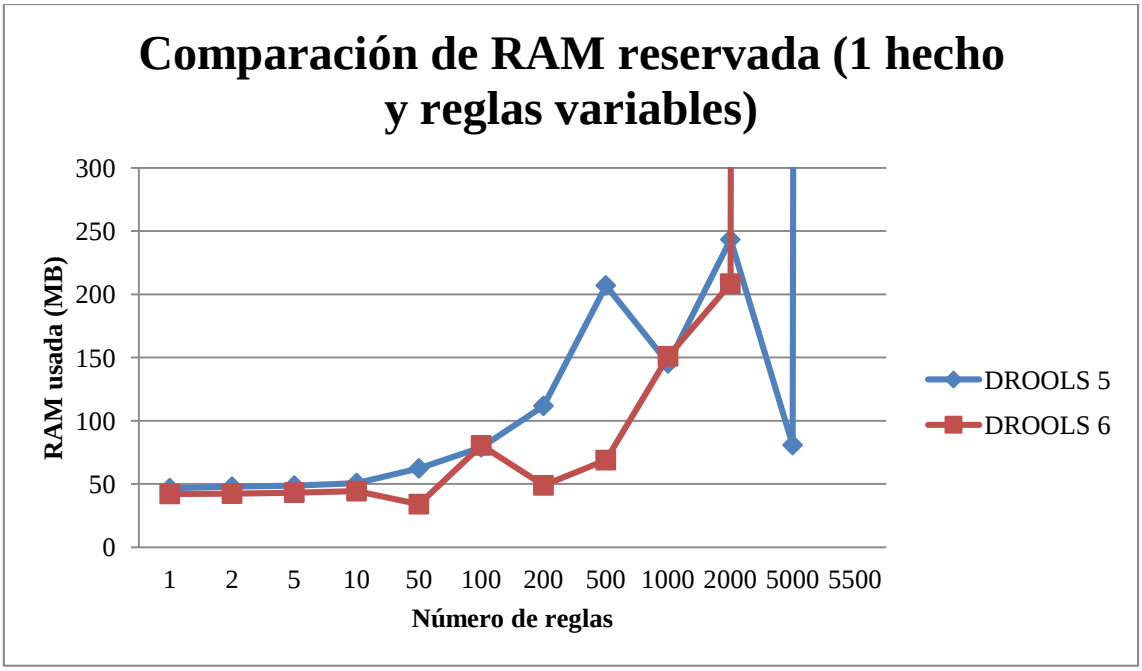

FIGURA 7: COMPARATIVA DE MEMORIA RESERVADA CON 1 REGLA Y HECHOS VARIABLES

Se ve que por normal general *DROOLS* 6 se sitúa por debajo aunque tampoco hay una gran diferencia. En el gráfico se ve que se admite un número máximo de reglas, y que a partir de ese máximo la memoria que ocupa ya es infinita puesto que la aplicación no sería capaz de terminar.

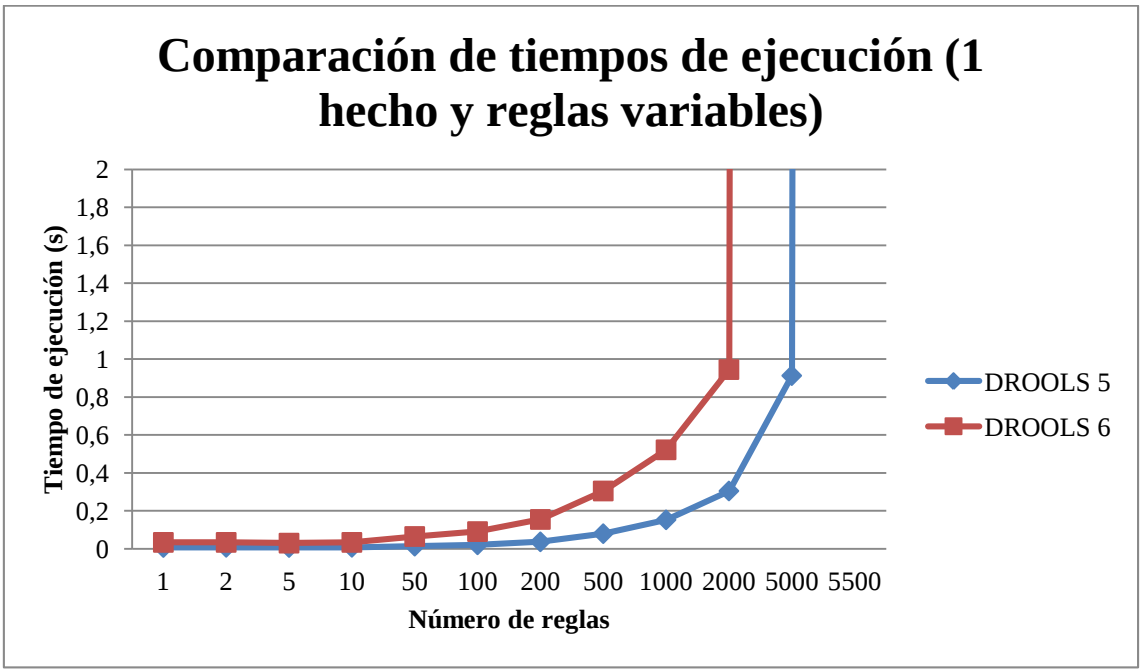

FIGURA 8: COMPARATIVA DE TIEMPOS DE EJECUCIÓN CON 1 HECHO Y REGLAS VARIABLES

Como caso curioso se puede observar que *DROOLS* 5 tarda menos tiempo en procesar un número de reglas mayor cuando sólo se tiene un hecho en la base.

A partir de ahora al ser BC y BH más grandes y se harán combinaciones de ambos, por lo que los datos de medición serán más precisos y diferirán más por lo que podemos obtener una comparativa más precisa de la escalabilidad.

Una aplicación con 100 reglas y cambiando el número de hechos:

100 Reglas / 100 Hechos	**CPU (%)**	**RAM (MB) Usada/Reservada**	**Tiempo (s)**
***DROOLS* 5**	0.00	87.67/340.25	0.429
***DROOLS* 6**	0.00	51.02/121.50	0.607

100 Reglas / 1000 Hechos	**CPU (%)**	**RAM (MB) Usada/Reservada**	**Tiempo (s)**
***DROOLS* 5**	0.00	159.75/340.25	0.684
***DROOLS* 6**	0.00	61.87/154.0	0.933

100 Reglas / 5000 Hechos	**CPU (%)**	**RAM (MB) Usada/Reservada**	**Tiempo (s)**
***DROOLS* 5**	18.84	194.75/340.43	1.537
***DROOLS* 6**	15.86	48.00/228.50	1.978

100 Reglas / 10000 Hechos	**CPU (%)**	**RAM (MB) Usada/Reservada**	**Tiempo (s)**
***DROOLS* 5**	17.32	379.78/495.375	2.499
***DROOLS* 6**	18.30	60.30/359.00	3.1

100 Reglas / 20000 Hechos	**CPU (%)**	**RAM (MB) Usada/Reservada**	**Tiempo (s)**

***DROOLS* 5**	18.58	364.55/739.81	3.889
***DROOLS* 6**	18.03	243.44/362.00	4.531

100 Reglas / 50000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	16.13	165.99/742.62	7.807
***DROOLS* 6**	16.86	167.65/442.50	7.814

100 Reglas / 100000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	15.41	556.88/607.25	14.57
***DROOLS* 6**	16.65	353.93/671.00	12.865

Se puede ver claramente que a medida que la base de hechos aumenta, *DROOLS* 6 es más eficiente en memoria y en tiempo respecto a *DROOLS* 5. La cantidad de CPU que se utiliza es similar en ambos casos.

Una aplicación de 500 reglas y cambiando el número de hechos:

500 Reglas / 100 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	0.00	249.31/343.75	0.625
***DROOLS* 6**	0.00	96.05/129.00	0.959

500 Reglas / 1000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	21.30	123.15/499.81	1.477
***DROOLS* 6**	18.09	88.08/215.50	1.989

500 Reglas / 5000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	19.59	51.15/742.87	5.073
***DROOLS* 6**	17.16	102.63/313.50	5.627

500 Reglas / 10000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	17.31	271.55/743.18	8.970
***DROOLS* 6**	21.93	222.14/272.50	11.36

500 Reglas / 20000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	18.24	139.695/576.312	16.263
***DROOLS* 6**	19.31	123.20/274.50	16.997

500 Reglas / 50000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)
***DROOLS* 5**	15.03	77.41/166.18	36.850
***DROOLS* 6**	16.34	69.19/154.00	34.493

500 Reglas / 100000 Hechos	CPU (%)	RAM (MB) Usada/Reservada	Tiempo (s)

***DROOLS* 5**	13.31	433.12/797.18	65.954
***DROOLS* 6**	15.20	702.28/783.50	56.876

Se ve que cuantos más datos se estén introduciendo, la mejora es mayor. Por último se analiza una aplicación de 2000 reglas y cambiando el número de hechos. En este último caso también se realizarán cálculos para ver la mejoría y se representará el estudio gráficamente. Por tanto para estas tablas se ha optado por usar un software que sea más preciso a la hora de calcular los datos[5].

2000 Reglas / 100 Hechos	**CPU (%)**	**RAM (MB) Virtual/Física**	**Tiempo (s)**
***DROOLS* 5**	16.73	789.15/539.34	0:00:19.640
***DROOLS* 6**	19.44	602.03/381.71	0:00:17.581

2000 Reglas / 1000 Hechos	**CPU (%)**	**RAM (MB) Virtual/Física**	**Tiempo (s)**
***DROOLS* 5**	12.64	926.36/760.36	0:00:25.459
***DROOLS* 6**	13.00	958.37/587.50	0:00:21.184

2000 Reglas / 5000 Hechos	**CPU (%)**	**RAM (MB) Virtual/Física**	**Tiempo (s)**
***DROOLS* 5**	12.40	933.35/787.03	0:00:48.251
***DROOLS* 6**	12.49	949.22/601.65	0:00:36.005

2000 Reglas / 10000 Hechos	**CPU (%)**	**RAM (MB) Virtual/Física**	**Tiempo (s)**
***DROOLS* 5**	12.51	929.55/849.19	0:01:22.727
***DROOLS* 6**	29.85	957.14/623.69	0:01:45.706

2000 Reglas / 20000 Hechos	**CPU (%)**	**RAM (MB) Virtual/Física**	**Tiempo (s)**
***DROOLS* 5**	12.44	924.29/871.75	0:02:29.355
***DROOLS* 6**	21.44	954.68/800.41	0:02:12.351

2000 Reglas / 50000 Hechos	**CPU (%)**	**RAM (MB) Virtual/Física**	**Tiempo (s)**
***DROOLS* 5**	12.55	954.37/913.06	0:05:08.476
***DROOLS* 6**	17.36	1062.10/896.14	0:02:52.303

2000 Reglas / 100000 Hechos	**CPU (%)**	**RAM (MB) Virtual/Física**	**Tiempo (s)**
***DROOLS* 5**	12.65	983.96/920.46	0:08:14.476
***DROOLS* 6**	16.08	1085.34/924.45	0:05:01.705

[5] Para esta medición haremos uso de un software externo, y tomaremos los datos desde la carga de las reglas hasta que termina el motor de inferencia. El software usado es el *process explorer* (http://technet.microsoft.com/es-es/sysinternals/bb896653.aspx) y se mide el pico de memoria física y virtual, el tiempo de ejecución desde que se cargan las reglas hasta que infiere la solución y el uso medio de CPU.

Con esto ya se tendría suficiente para hacer el estudio de escalabilidad, puesto que bases más grandes no serían realistas y en rara ocasión tendría lugar en la vida real.

Como conclusión final se puede decir que la última versión de *DROOLS* usa más CPU para funcionar pero a cambio consigue que la aplicación finalice en bastante menos tiempo cuando las BH y BC superan un umbral. En cuanto al tema de memoria, usada y la memoria física también se mejora pero no es tanta la mejoría. Por lo general, es un cambio bastante significativo el usar un algoritmo distinto al clásico *RETE* que se lleva usando desde el principio.

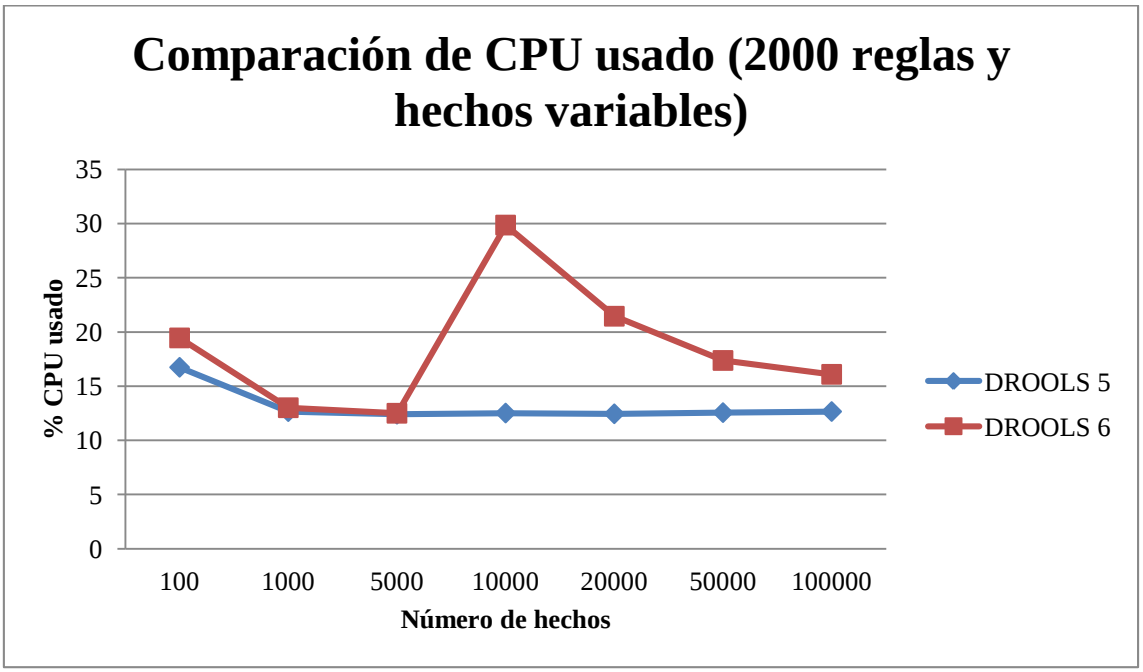

FIGURA 9: COMPARATIVA DE CPU USADA CON 2000 REGLAS Y HECHOS VARIABLES

En media, *DROOLS* 5 usa un 13.13% de CPU mientras que *DROOLS 6* usa un 18.52%. Por lo tanto se puede concluir que *DROOLS* 6 usa un 41% más de CPU que *DROOLS 5*.

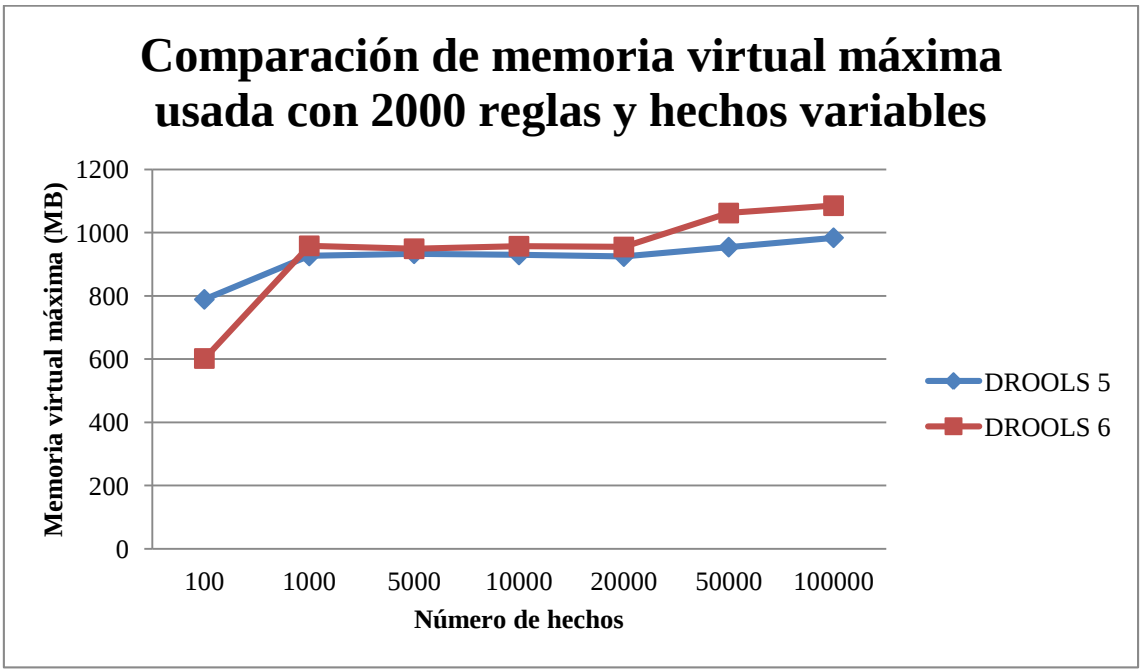

FIGURA 10: COMPARATIVA DE MEMORIA VIRTUAL MÁXIMA USADA CON 2000 REGLAS Y HECHOS VARIABLES

DROOLS 5 usa de media de memoria virtual máxima de 920.14 MB y *DROOLS* 6 un poco más, 938.41 MB. La diferencia no es mucha y es más importante la memoria física que utilice que se representa en el siguiente gráfico.

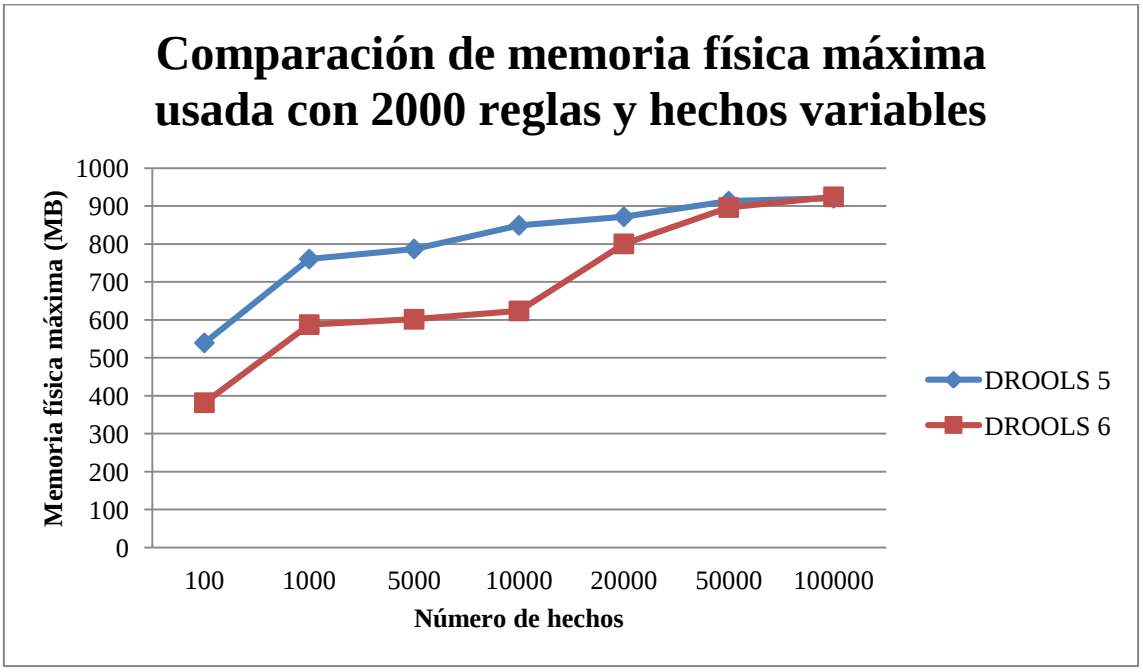

FIGURA 11: COMPARATIVA DE MEMORIA FÍSICA MÁXIMA USADA CON 2000 REGLAS Y HECHOS VARIABLES

Se puede ver directamente que *DROOLS* 6 está por debajo en la mayoría de casos hasta que ambos convergen. Cabe destacar que rara vez una aplicación podría llevar tantos hechos. De media, *DROOLS* 5 usa de máximo 805.88 MB mientras que *DROOLS* 6 tiene de media 687.93

MB. Por tanto la última versión de *DROOLS* ahorra aproximadamente un 15% de memoria física en sus ejecuciones.

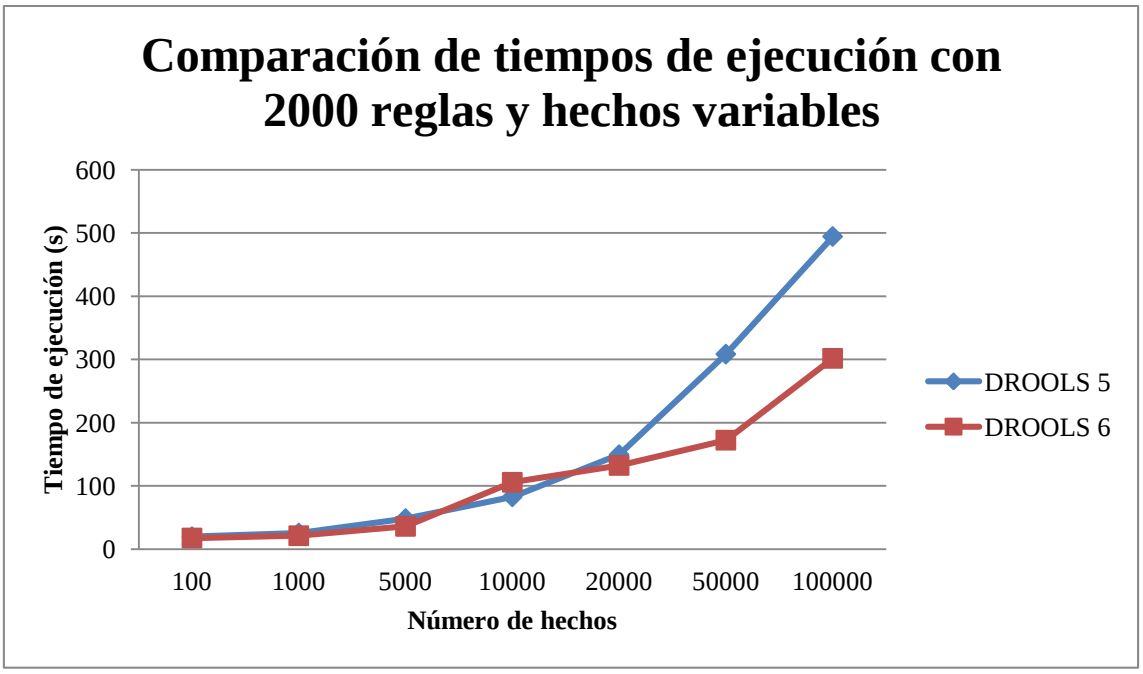

FIGURA 12: COMPARATIVA EN TIEMPOS DE EJECUCIÓN CON 2000 REGLAS Y HECHOS VARIABLES

Lo más significativo aquí es que en cuestión de tiempos, *DROOLS* 6 es mucho más escalable. A partir de un tamaño grande, el tiempo de ejecución es mucho menor en esta última versión, aspecto que en la mayoría de casos es la que más importancia tiene. En media, *DROOLS* 5 se ejecuta en 161.19s mientras que *DROOLS* 6 tarda de media 112.4s siendo un 43% más rápido que su predecesor.

3.1.1.2. CROSS PRODUCTS Y QUERIES

En *DROOLS* existe un término llamado *Cross Product* que funciona como un *join* (mencionado anteriormente en las descripciones de los algoritmos) de SQL. Esto se trata simplemente se realizar un producto cruzado cuando se tienen creados varios objetos y están como antecedentes en una regla. Si se hace buen uso de los *Cross Product* se ahorrará bastante trabajo así como memoria, puesto que se evitaría generar datos inútiles que lo único que hacen es que el motor tarde más en terminar de inferir y ocupar espacio.

Para que se entienda mejor se presenta un ejemplo:

- Sean dos tipos de clases: `Contenedor` y `Contenido`.
- La clase `Contenido` pertenece a un `Contenedor`. Por lo tanto tiene un atributo `Contenedor` que indica cuál es su `Contenedor`.
- Se tienen cuatro objetos de cada tipo: `contenedorA`, `contenedorB`, `contenedorC`, `contenidoA`, `contenidoB` y `contenidoD` (cada una asociada a su contenedor correspondiente).

A la hora de definir las reglas, se deberá de tener en cuenta qué datos son los que interesan ser comparados y guardados. Por ejemplo la regla:

```
rule "Mostrar contenido" when
```

```
      $contenedor: Contenedor()

      $contenido: Contenido()
then

      System.out.println( “contenedor:” + $contenedor.getNombre()
+ “ contenido:” + $contenido.getNombre() );

end
```

Al no especificar ninguna restricción, se hará un *cross product* de todos los objetos, y devolverá 9 parejas de datos, de los cuales la mayoría no interesan pues no coincide el `Contenido` con su `Contenedor`. Lo que devuelve en este caso es:

```
Contenedor: contenedorA Contenido: contenidoA
Contenedor: contenedorA Contenido: contenidoB
Contenedor: contenedorA Contenido: contenidoC
Contenedor: contenedorB Contenido: contenidoA
Contenedor: contenedorB Contenido: contenidoB
Contenedor: contenedorB Contenido: contenidoC
Contenedor: contenedorC Contenido: contenidoA
Contenedor: contenedorC Contenido: contenidoB
Contenedor: contenedorC Contenido: contenidoC
```

De este resultado sólo interesan las parejas que coinciden, por lo que tenemos que restringir el *cross product* para que sea más eficiente en cuanto a tiempo y memoria. Si no se especificasen correctamente, para reglas que utilicen muchos objetos, el *cross product* podría ser demasiado grande, causando que el motor de inferencias sea extremadamente lento cuando no hay necesidad de ello. La regla correcta sería:

```
rule “Mostrar contenido” when

      $contenedor: Contenedor()

      $contenido: Contenido( contenedor == $contenedor)
then

      System.out.println( “contenedor:” + $contenedor.getNombre()
+ “ contenido:” + $contenido.getNombre() );

end
```

De esta manera nos aseguramos que los contenidos que se muestren sean sólo aquellos que se asocien de manera correcta a su contenedor. El resultado en este caso sería:

```
Contenedor: contenedorA Contenido: contenidoA
Contenedor: contenedorB Contenido: contenidoB
Contenedor: contenedorC Contenido: contenidoC
```

En este ejemplo tan sencillo ya se está utilizando 2/3 menos de memoria, lo cual de cara a una aplicación grande con muchas reglas y hechos podría traducirse a algo más grande.

En cuanto a las *queries* se trata de un tipo de estructura integrado en *DROOLS* que se puede especificar en el fichero `.drl`. Es una manera más sencilla de buscar en la memoria de trabajo hechos que coincidan con unas condiciones dadas. Por lo tanto, sólo contiene la parte derecha

de una regla o en este caso la parte que se especifica en el `when` obviando poner el `then` y la parte que le sigue. En una *query* también se tiene una serie de parámetros que son opcionales; por ejemplo si no se pone el tipo del objeto, se asume que es el tipo *object* y el motor intentará asociarle el valor que considere que es. Además los nombres de las *queries* son globales para toda la *KieBase* por lo que no se pueden añadir *queries* del mismo nombre a diferentes paquetes para una misma *RuleBase*.

Si se quiere hacer una consulta, únicamente se escribirá `kSession.getQueryResults("nombre")` donde "nombre" es el nombre de la *query*. Esto devolverá la lista de resultados de la *query* lo que permite recuperar los objetos que coinciden con la misma. Por ejemplo, si se tiene una BC de personas con un atributo edad, y se quieren recuperar todas las personas con una edad mayor de 30, se escribiría la siguiente *query*:

```
query "personas mayores de 30"

     persona: Persona ( edad > 30 )

end
```

Aplicando el método anteriormente explicado, tendremos una lista de personas que cumpla esa *query*. También se puede parametrizar la *query* para que cumplan ciertas condiciones. Por ejemplo si se quiere obtener la lista de personas de una edad superior a `x` que vivan en `y` se escribirá:

```
query "mayores de x que viven en y" (int x, String y)

     persona: Persona ( edad > x, localidad == y )

end
```

Esta lista que se obtiene al consultar la *query* se puede recorrer usando un iterador *for*. Cada elemento de la lista es un `QueryResultsRow` y se puede acceder a los datos de la misma que estará en forma de columnas (cada fila es un resultado y cada columna de la misma los datos). El acceso a las columnas se puede hacer por índice o por nombre.

El código Java equivalente a realizar una consulta sobre las personas mayores de 30 años sería el siguiente:

```
QueryResults resultados = kSession.getQueryResults("personas
mayores de 30");

System.out.println("Tenemos " + resultados.size() + " personas
mayores de 30 años");

System.out.println("Estas son las personas mayores de 30:");

for (QueryResultsRow fila : resultados){

     Persona persona = (Persona)fila.get("persona");

     System.out.println("persona.getNombre());

}
```

Además para hacer el código más compacto, se añade una sintaxis posicional. Por defecto, cuando se declara un tipo de objeto con varios atributos, la posición de cada uno será en función al orden de declaración. No obstante podemos modificarla usando la anotación `@position`. En un ejemplo se puede ver más claro:

```
declare Queso

      nombre: String @position(1)

      tienda: String @position(2)

      precio: int @position(0)

end
```

Si no se hubiera usado la anotación `@position` entonces nombre sería la 0, tienda la 1 y precio la 2. De esta manera la *query* `isContainedIn` que sirve para comprobar si un objeto está contenido en una localización pasará de tener el patrón `Location(x, y;)` en vez de `Location(objeto == x, localización == y)`.

De esta manera ahora se puede llamar desde una *query* a otra *query* que combinado con algunos argumentos opcionales nos permite una realizar *queries* como una derivación del mecanismo de encadenamiento hacia atrás. Un ejemplo con una *query* que se llama a si misma sería el siguiente:

```
query isContainedIn(String x, String y)

      Location(x, y;)

      or

      (Location (z, y;) and isContainedIn(x, z;))

end
```

De esta manera se devolverían aquellas `Location` que cumplen que tienen un objeto `x` y una localización `y` o bien que tienen una localización `y`, pero el objeto que tiene, es una localización de otro `Location` que tiene como objeto la `x` pasada como argumento.

3.1.1.3. COMPARATIVA EN CONSULTA DE *FACTS*

Para este apartado se va a crear estructuras de datos nuevas acerca de individuos y haremos *queries* para consultar los datos de los mismos y calcular el tiempo que tarda *DROOLS* 5 y *DROOLS* 6 en resolver la consulta.

Se va a hacer uso de una clase `Persona` que tiene como atributos un entero que será la `edad` y un atributo que será una cadena que represente el `nombre`. En el fichero de *DROOLS* tendremos una *query* que permitirá buscar en la BH aquellas personas que cumplan una condición de edad y de nombre. En la BH inicial se introducirán una serie de personas, y sólo 1/3 de ellas cumplirán esas condiciones (de esta manera también se puede ver qué versión de *DROOLS* es más eficiente a la hora de comparar y descartar hechos).

Se medirá la RAM máxima que usa en la ejecución (tanto física como virtual) y el tiempo que tarda en devolver la consulta. Estas mediciones se harán con las mismas herramientas que se usó

para evaluar el tiempo de inferencia dado un determinado número de reglas y hechos que se analizó en el apartado anterior. Los datos que se obtienen son los siguientes:

300 Hechos	RAM (MB) Virtual/Física	Tiempo (s)
***DROOLS* 5**	414.72/181.01	0.020
***DROOLS* 6**	307.85/90.44	0.090

3000 Hechos	RAM (MB) Virtual/Física	Tiempo (s)
***DROOLS* 5**	414.00/186.69	0.490
***DROOLS* 6**	308.00/90.69	0.185

15000 Hechos	RAM (MB) Virtual/Física	Tiempo (s)
***DROOLS* 5**	434.71/234.97	1.293
***DROOLS* 6**	311.12/106.68	0.449

30000 Hechos	RAM (MB) Virtual/Física	Tiempo (s)
***DROOLS* 5**	582.13/332.08	1.776
***DROOLS* 6**	308.74/109.43	0.519

60000 Hechos	RAM (MB) Virtual/Física	Tiempo (s)
***DROOLS* 5**	583.75/337.87	2.298
***DROOLS* 6**	311.17/111.38	0.568

150000 Hechos	RAM (MB) Virtual/Física	Tiempo (s)
***DROOLS* 5**	838.03/523.52	4.132
***DROOLS* 6**	308.60/142.06	0.678

300000 Hechos	RAM (MB) Virtual – Física	Tiempo (s)
***DROOLS* 5**	834.30/782.05	7.038
***DROOLS* 6**	654.43/263.01	1.577

600000 Hechos	RAM (MB) Virtual/Física	Tiempo (s)
***DROOLS* 5**	992.82/862.54	12.327
***DROOLS* 6**	706.39/347.89	1.791

1500000 Hechos	RAM (MB) Virtual/Física	Tiempo (s)
***DROOLS* 5**	1298.82/1093.23	30.662
***DROOLS* 6**	1378.72/1074.31	6.395

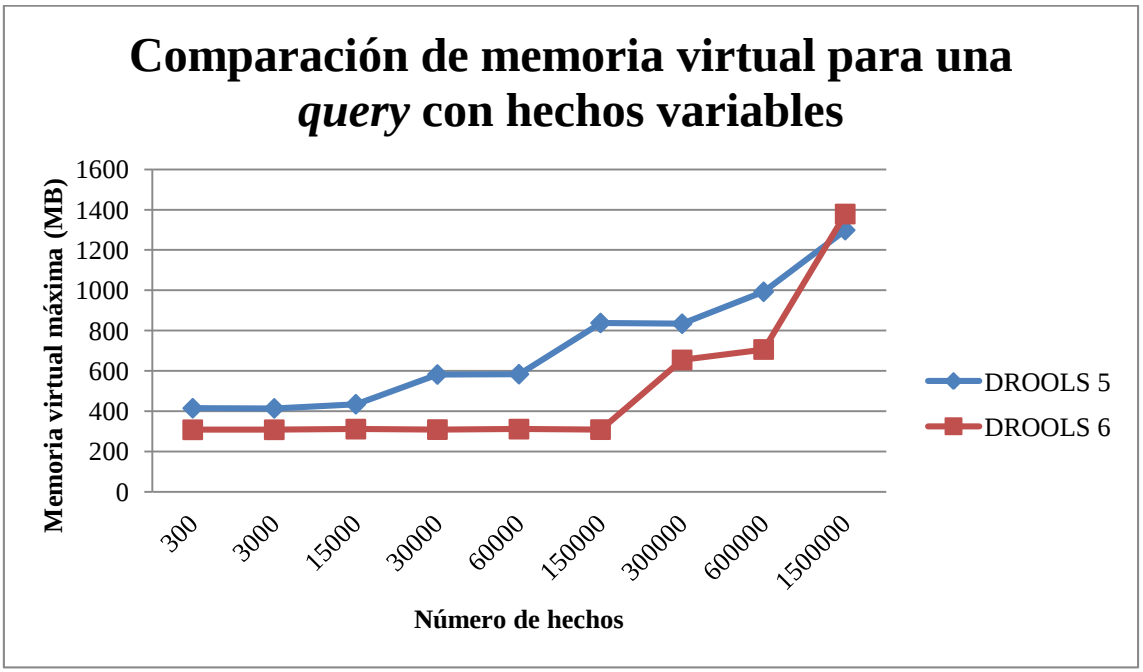

FIGURA 13: COMPARATIVA DE MEMORIA VIRTUAL MÁXIMA USADA PARA UNA *QUERY* CON HECHOS VARIABLES

Se ve como curiosidad que *DROOLS* 6 reserva mucha menos memoria virtual cuando son pocos hechos, pero al final a medida que el sistema tiene más hechos crece mucho más rápido, y puede ser debido a que el sistema tiene un máximo de memoria virtual permitido y en esos casos ambos se sitúan al límite. De media *DROOLS* 5 tiene un pico de memoria virtual de 710.36 MB mientras que *DROOLS* 6 tiene 510.55 MB por lo que ocupa un 39% menos.

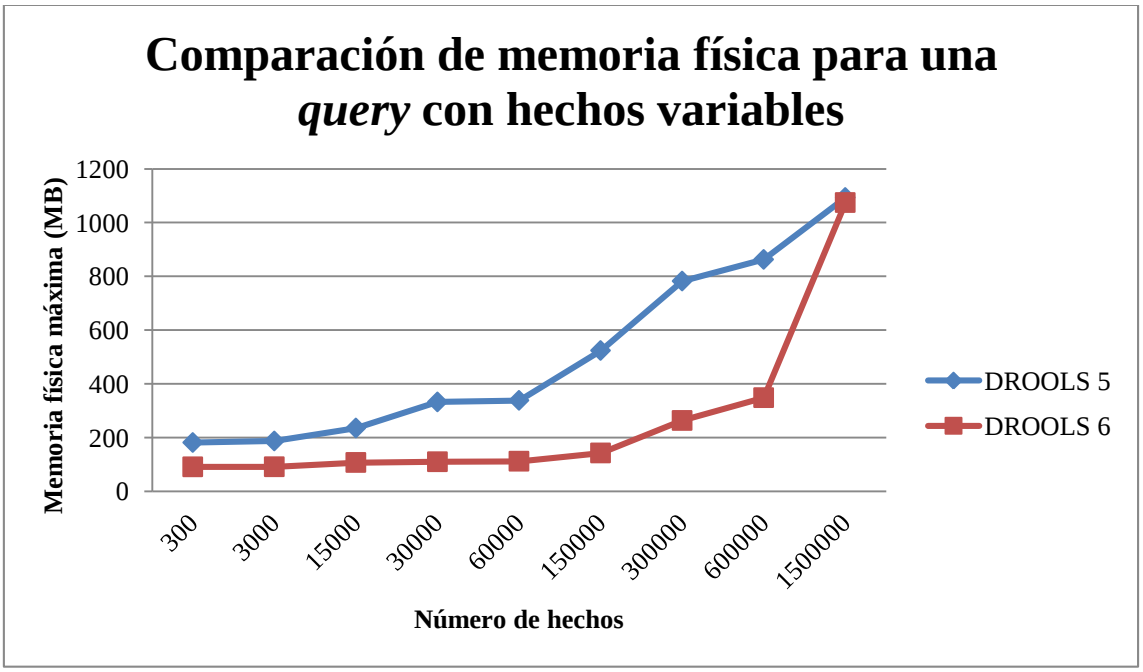

FIGURA 14: COMPARATIVA DE MEMORIA FÍSICA MÁXIMA USADA PARA UNA *QUERY* CON HECHOS VARIABLES

En cuanto a memoria física, parte que se considera más importante porque es la parte limitada de un ordenador se ve que *DROOLS* 6 está bastante más optimizado, aunque al igual que con la

memoria virtual, llega un punto en el que ambos convergen por no disponer de más memoria disponible para la aplicación. De media *DROOLS* 5 ocupa un pico de 503.77 MB y *DROOLS* 6 tan sólo 259.54 MB por lo que ocupa un 94% menos, que es casi la mitad.

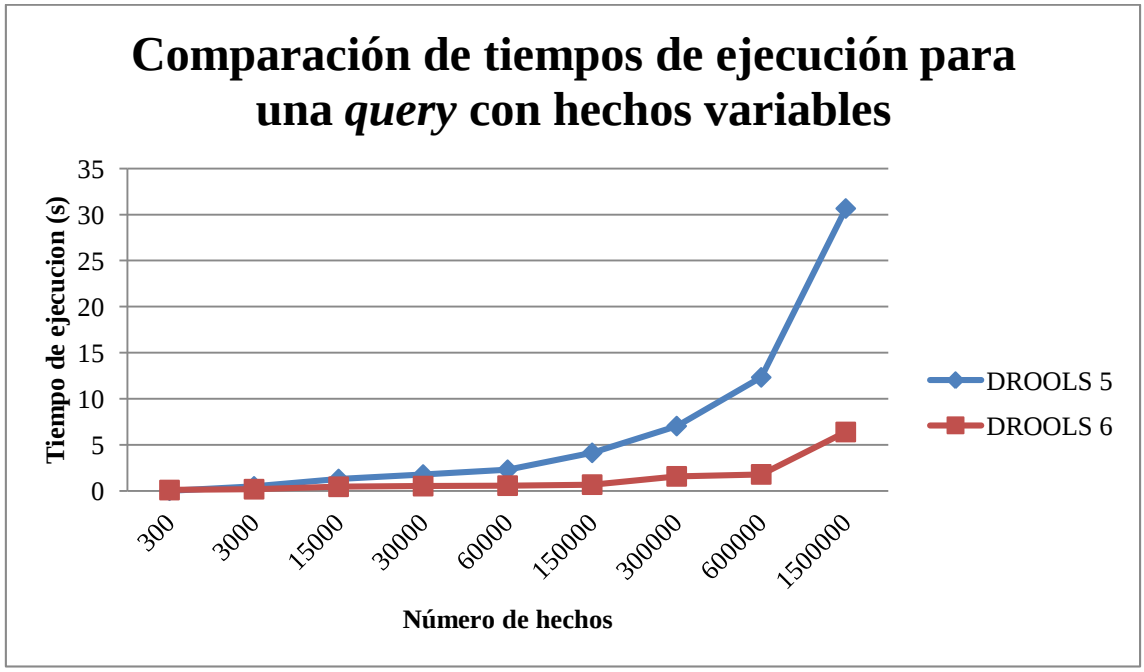

FIGURA 15: COMPARATIVA DE TIEMPOS DE EJECUCIÓN PARA UNA *QUERY* CON HECHOS VARIABLES

Al igual que con el motor de inferencias, donde más ventaja saca la última versión de *DROOLS* es en cuanto a tiempo. Se ve claramente que es mucho más eficiente en cuanto a tiempo las *queries* en *DROOLS* 6. El tiempo promedio de ejecución en estas pruebas en *DROOLS* 5 es de 6.67s y el de *DROOLS* 6 es de 1.36s por lo que es un 390% más rápido.

3.1.2. Diagrama de clases
3.1.3. Diagrama de paquetes
3.1.4. Ejemplo de aplicación

3.2. La aplicación en *OpenCDS*

3.2.1. Diagrama de clases
3.2.2. Diagrama de paquetes
3.2.3. Etapas de desarrollo
3.2.4. Requisitos
3.2.5. Casos de uso
3.2.6. Implementación y desarrollo
3.2.7. Instrucciones de uso

3.2.8. SCREENSHOTS

4. CONCLUSIONES

ANEXO I: INSTALACIÓN Y CONFIGURACIÓN DE *OPENCDS*

ANEXO II: CREACIÓN DE UN PROYECTO PEQUEÑO EN *OPENCDS*

GLOSARIO DE TÉRMINOS

- **BC**: Base de Conocimiento
- **BH**: Base de Hechos
- **SBR**: Sistema Basado en Reglas
- **SNS**: Servicio Nacional de Salud

BIBLIOGRAFÍA Y REFERENCIAS

[1] Mark Proctor. R.I.P RETE time to get PHREAKY. 1 de Noviembre de 2013, http://blog.athico.com/2013/11/rip-RETE-time-to-get-phreaky.html.

[2] The JBoss Drools Team (2013). *Drools Documentation* (Version 6.0.1.Final), Cross Products (pp. 144-145), Queries (pp. 283-286).

[3] José T. Palma (2010). *Sistemas Basados en Reglas: Técnicas de Equiparación.* Desarrollo de Sistemas Inteligentes, DIIC, UMU.