UNIVERSIDAD DE LA REPÚBLICA

Facultad de Ingeniería

InCo - Instituto de Computación

"Diseño e implementación de un motor de reglas dinámicas usando especificaciones GeneXus"

Tesis de Maestría presentada por

Luciano Silveira

Tutor

M.Sc. Juan José Prada

Informe

Montevideo, Uruguay - 2010

Resumen

Las organizaciones poseen un conjunto de políticas que utilizan como fundamento para operar. Por lo general las reglas de negocio definen estas políticas, las prácticas y los procedimientos que se deben de llevar a cabo para satisfacer los objetivos del negocio.

Las reglas de negocio no son estáticas; las empresas que continuamente ajusten sus procesos y reglas en respuesta a las condiciones de negocio cambiantes estarán mejor preparadas para afrontar las demandas del mercado y expectativas de los clientes.

En los últimos tiempos, ha tomado fuerza la idea de diseñar y administrar estas restricciones y políticas (representadas como reglas de negocio) en forma independiente de los procesos del negocio y aplicaciones, de tal forma que los cambios en restricciones, reglas y políticas empresariales puedan ser especificados en forma independiente por un analista del negocio sin necesidad de modificar las aplicaciones.

Un motor de evaluación de reglas de negocio provee capacidades de este estilo, donde las políticas empresariales se declaran a un nivel de abstracción acorde al analista de negocio. Un motor toma estas definiciones y a partir de los datos recibidos, ejecuta las acciones que indiquen las reglas. Se logra una mayor agilidad y transparencia llevando a las empresas a mejorar la modularidad y accesibilidad de sus reglas y políticas empresariales, al separar éstas de los procesos empresariales y de la lógica de presentación.

Desde sus orígenes, GeneXus ha tomado como fundamento la representación abstracta de la realidad, la utilización de reglas de negocios en forma declarativa, y el automatismo para la generación de soluciones. Estos son los pilares sobre los que se basa para acompasar el permanente proceso de cambio tecnológico y de negocios.

Este trabajo desarrolla un prototipo en donde las reglas de negocio pasan a ser un objeto en sí mismo dentro del entorno de desarrollo GeneXus. Aquellas reglas de negocio que tengan un alto impacto en la toma de decisiones y sean factibles de cambiar frecuentemente, se administran en forma externa a la aplicación debido a que las principales funcionalidades de una aplicación no necesariamente cambian al mismo ritmo que los procesos de negocio.

Se diseña un lenguaje de dominio específico personalizable a nivel del experto del negocio para facilitar el entendimiento y que el experto tenga la posibilidad de modificar dichas reglas. Las reglas de negocio se administran en forma independiente de la aplicación, y se proporciona un editor de reglas para que el experto del negocio

tenga la posibilidad de realizar las modificaciones necesarias en el propio entorno de ejecución.

Además, se utilizan motores de evaluación de reglas de negocio existentes como mecanismo de ejecución de las reglas definas.

Palabras clave: *regla de negocio, GeneXus, lenguaje de dominio específico.*

Abstract

Organizations usually define a set of policies which are used as a basis to run their business. Business rules define these policies, the procedures and practices that must be carried out to meet the business objectives.

Business rules are not static; those enterprises which continually adjust their processes and rules as a response to changing business conditions will be better prepared to meet market demands and customer expectations.

In recent times, the idea of designing and managing these restrictions and policies (modeled as business rules) independently from business processes and applications core functionality has gained strength. This means that changes in restrictions, regulations and corporate policies can be specified in an independent way by a business expert without the need to modify existing applications.

A business rules engine provides these capabilities, in which corporate policies are declared at an abstraction level specification depending on the business analyst. A dedicated rule engine takes these definitions as input to execute the actions specified by the high-level rules. Consequently, greater agility and transparency is achieved, thus leading companies to an improvement of modularity and accessibility of its business rules and policies by decoupling these restrictions from the existing processes and presentation logic.

Since its conception, GeneXus has based its approach on an abstract representation of reality, on the use of business rules declaratively, and the automation to generate business solutions. These are the foundations on which GeneXus is based to keep abreast with the continuous process of technological and business change.

This dissertation develops a prototype in which the business rules become an object per se within the GeneXus Development Environment. Those business rules, which have a high impact on decision making and which may change frequently will be managed outside the application because the core application functionality may not necessarily change at the same pace as the business processes.

A dedicated high-level business rule language is designed to enable the business expert to understand and define high-level rules. This decouples the business rules from the application core and a rule designer is provided to enable the business expert to make the necessary changes in the execution environment.

In addition, existing business rules engines are used as a framework for executing the defined business rules.

Keywords: *business rule, GeneXus, domain-specific language.*

Índice general

Índice de figuras

1. Introducción

Este trabajo plantea estudiar las ideas fundamentales que propone el enfoque de uso de reglas de negocio, los motores de evaluación de reglas y gerenciadores de reglas de negocio. Se explora la aplicabilidad de los conceptos en el contexto GeneXus con el objetivo de lograr aplicaciones flexibles con altos niveles de parametrización, con el fin de mejorar la agilidad de los procesos empresariales.

1.1. Motivaciones y problemas

Las organizaciones poseen un conjunto de reglas y políticas empresariales que utilizan como fundamento para operar. Por lo general, las reglas de negocio definen éstas políticas, las prácticas y los procedimientos que se deben llevar a cabo para satisfacer los objetivos del negocio. Normalmente, ningún individuo conoce dichas restricciones en su totalidad, sino que seguramente las conozca en forma parcial. También ocurre que la lógica que satisface dichas necesidades no se encuentra completamente formalizada, por lo que al producirse cambios en el negocio, no se reflejan dichos cambios en éstas definiciones, generando inconsistencias en el largo plazo.

Es fundamental que todas las aplicaciones de negocio se rijan por dichas reglas, ya que estas reglas de negocio forman los cimientos de la toma de decisiones en una empresa. Si bien al diseñar las reglas de negocio se toman en cuenta dichas consideraciones, en muchos casos las reglas quedan ocultas dentro del código de la aplicación que implementa la funcionalidad requerida. A medida que la realidad cambia y se realizan ajustes, es usual que se pierda el contexto de cómo fue implementada la regla de negocio a nivel del código fuente.

Tanto reglas, restricciones, políticas empresariales como puntos de decisión importantes quedan embebidos dentro de múltiples procesos o procedimientos, no quedando claro qué proceso depende de cuál política o regla. Los analistas de negocio pierden visibilidad y no son capaces de administrar dichas reglas, encontrándose en muchos casos, versiones inconsistentes de lo que deberían ser las mismas reglas en diferentes ubicaciones. Esto genera inconsistencias en los procesos de negocio, por lo que es deseable que esta lógica sea extraída, formalizada, y administrada en forma centralizada y además se garantice su cumplimiento en toda la organización.

Otro punto a considerar, es la restricción en relación al tiempo disponible para implementar los cambios en el negocio. Facilitar la modificación de políticas

empresariales y permanecer competitivos en el contexto actual, implica poder reflejar en forma eficiente y efectiva cualquier cambio en el negocio. Utilizando el enfoque tradicional de desarrollo de software, en donde muchas de las restricciones y políticas que comandan el negocio se encuentran embebidas en las aplicaciones, implica realizar una reprogramación para lograr el cambio deseado.

Desde hace algunos años ha tomado fuerza la idea de diseñar y administrar estas restricciones y políticas (representadas como reglas de negocio) en forma independiente de los procesos del sistema. El enfoque de reglas de negocio (BRA - Business Rules Approach) [204] separa los procesos que lleva a cabo la empresa de las restricciones que se deben cumplir – de tal forma que cambios en restricciones y políticas empresariales, puedan ser especificados por un analista del negocio no necesariamente un profesional informático – sin necesidad de reprogramar la aplicación para modelar nuevas restricciones.

El enfoque de reglas de negocio plantea administrar este tipo de reglas en forma independiente de la aplicación, separando cierta parte de la lógica de negocio en forma externa al proceso en sí – especificados como reglas – de manera que tanto la aplicación como las reglas puedan evolucionar en forma independiente.

Un uso adecuando de este tipo de tecnología tiene el potencial de crear aplicaciones realmente dinámicas, flexibles y adaptables al cambio. Las empresas que continuamente ajusten sus procesos y reglas en respuesta a las condiciones de negocio cambiantes, estarán mejor preparadas para afrontar futuros cambios.

1.2. Propuesta

Cuando se diseña una aplicación, es de gran importancia conceptualizar cuál es la lógica del negocio. La presentación de una aplicación cambia con las diferentes tecnologías, pero las reglas del negocio permanecen siendo un patrimonio vital del negocio. En la medida que se pueda efectuar esta separación, se estará mejor preparado para afrontar las demandas del mercado y expectativas de los clientes.

Una de las posibles soluciones para lograr esta separación implica aislar lo máximo posible de la lógica de negocio y en particular de la programación, a aquellas restricciones clave que rigen al negocio. También, es deseable que dichas restricciones se administren al mayor nivel de abstracción para que las mismas sean comprendidas por toda la organización, en particular por los expertos del negocio.

Para fijar ideas, supongamos un escenario tipo de facturación, donde se desea que la aplicación tenga la capacidad de cambiar cierto comportamiento dependiendo de diferentes parámetros del sistema, como por ejemplo poder diferenciar los precios de ciertos productos, aplicar descuentos o beneficios, así como también presentar promociones especiales en diferentes períodos de tiempo a un grupo selecto de clientes.

Una regla de negocio que modele un descuento podría ser la siguiente:

```
Todos los clientes
  con categoría "A" asignada,
  que sean de Montevideo,
  que tengan órdenes de compras en Enero 2009,
reciben un 10% de descuento.
```

En vez de implementar dicha restricción con código procedural, las reglas de negocio definen declaraciones del tipo "*If <condition> then <action>*", que se corresponden con las políticas definidas por la organización, son declarativas y no procedurales y serán procesadas por un motor de evaluación de reglas. En términos generales, se dividen en dos bloques:

- Sección *If*: se le considera la condición de la regla.
- Sección *Then*: es la parte de acciones o consecuencia de la regla.

donde la acción es ejecutada cuando la condición se cumple. Siguiendo el ejemplo, una representación de la regla en pseudo-código se detalla a continuación:

```
rule "Promoción Especial Enero 2009"
  If
    Cliente.Departamento = "MVD"
    Cliente.Categoria = "A"
    Orden.Fecha >= '20090101'
    Orden.Fecha <= '20090131'
  then
    Cliente.AsignarDescuento(10)
    EnviarMensaje("Descuento del 10% asignado")
end
```

En este caso se está indicando qué hacer en vez de cómo hacerlo si lo comparamos con el enfoque procedural; es decir, las reglas nunca se llaman directamente, éstas se ejecutan automáticamente por un componente de evaluación de reglas dependiendo de las condiciones que la compongan. En este caso, al procesar una factura, si se encuentra en el período de tiempo definido y además el cliente cumple todas las restricciones, se aplican las acciones correspondientes, que constan de aplicar un descuento y enviar un mensaje para informar de la situación.

Es en este contexto, donde el enfoque de reglas de negocio cumple un rol fundamental. Las reglas y políticas empresariales (restricciones) se separan de los procesos de negocio que lleva a cabo la empresa, permitiendo modificar dichas restricciones sin tener que modificar en forma explícita los procesos que las implementan.

Por su parte, GeneXus [178] provee un componente de definición de reglas de negocio declarativo, en donde las sentencias se especifican sin tener en cuenta el flujo de control. Sin embargo, es necesario realizar un proceso de generación de código

donde se analizan las dependencias existentes y se traduce a código procedural dichas dependencias. Cualquier cambio que se realice sobre las reglas de negocio, implica la regeneración de los programas fuentes.

Los motores de evaluación de reglas (BRE - Business Rules Engine) [135] adquieren particular relevancia en este contexto debido a que ofrecen capacidades para que los analistas de negocio realicen cambios dinámicos en tiempo real en sus aplicaciones, usando un nivel de lenguaje abstracto para administrar las reglas de negocio. Se propone investigar el uso de dichas tecnologías en el ecosistema GeneXus para lograr un manejo dinámico de las reglas de negocio y así lograr el desarrollo de componentes flexibles, fácilmente modificables y adaptables al cambio.

Algunos beneficios que se obtienen al utilizar dicha tecnología son:

- *Agilidad*: respuesta simple y rápida al cambio. Se modifica una regla en un único lugar e impacta en forma automática en aquellos lugares donde se utilice.
- *Reducción en costos*: se reduce sensiblemente los costos involucrados al agregar una nueva restricción utilizando el mecanismo de evaluación de reglas versus la programación procedural y posterior puesta en producción.
- *Transparencia*: un auditor o analista puede analizar el código y fácilmente determinar que la aplicación implementa en forma precisa las políticas y procedimientos del negocio.
- *Colaboración*: la administración de las reglas, a un nivel de abstracción acorde al experto del negocio, aumenta la colaboración con el departamento de Tecnologías de la Información ya que se cuenta con un lenguaje común de comunicación.

Por lo tanto, dentro del contexto Genexus, en este trabajo se estudia la interacción de una aplicación GeneXus con un motor de evaluación de reglas de manera de obtener los beneficios detallados anteriormente.

1.3. Objetivos

Los objetivos del proyecto son diversos. Inicialmente, comprender la propuesta de los Motores de Reglas (BRE - Business Rule Engines) y Gerenciadores de Reglas (BRMS - Business Rule Management System) [221] analizando tanto la teoría como productos disponibles en el mercado y brindar una síntesis que describa el tipo de problemática que resuelve.

Luego, realizar una experiencia práctica de estos conceptos aplicándolos en el desarrollo de una aplicación GeneXus, incorporando características para administrar las reglas de negocio en forma independiente a la plataforma. En este sentido, diseñar herramientas que permitan construir aplicaciones utilizando el concepto de reglas de

negocio; es decir, representar en forma declarativa aquellas restricciones y políticas clave para la organización, y fundamentalmente facilitar su rápida modificación a fin de responder ante cambios en el entorno empresarial.

En tercer lugar, construir un prototipo con el cual se estudien las dificultades a superar para implementar el concepto de reglas de negocio dinámicas que puedan ser administradas por expertos del negocio y ensayar posibles soluciones.

Finalmente, evaluar beneficios y limitaciones del enfoque de reglas de negocio dentro del contexto de los Sistemas de Información para la plataforma GeneXus.

1.4. Cronograma

A lo largo del proyecto se siguió un modelo de proceso iterativo e incremental. A grandes rasgos, las fases principales llevadas a cabo fueron las siguientes:

i. Estudio inicial del enfoque.

ii. Estudio bibliográfico y herramientas relacionadas.

iii. Revisión del estado del arte.

iv. GeneXus y el enfoque de reglas de negocio.

v. Especificación de requerimientos y propuesta.

vi. Experimentación con motores de reglas de negocio en diferentes plataformas.

vii. Diseño de la solución.

viii. Desarrollo del prototipo.

ix. Lenguajes de definición de reglas de alto nivel de abstracción y lenguajes específicos de dominio.

x. Diseño y desarrollo de un prototipo que incorpora lenguajes de reglas de alto nivel de abstracción.

xi. Integración de los artefactos utilizados para administrar las reglas de negocio en un prototipo integrado.

xii. Hitos principales relacionados con la documentación del trabajo.

xiii. Preparación del Informe.

Mes	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
i	x	x																	
ii			x	x	x														
iii						x													
iv						x	x												
v							x	x											
vi								x	x										
vii								x	x	x									
viii										x	x								
xix												x							
x												x	x	x	x	x			
xi																x	x		
xii			x			x			x			x			x		x		
xiii																x	x	x	x

La fase 1 comprende el entendimiento del problema, realizando un análisis inicial del enfoque: reglas de negocio.

En la fase 2 se estudia a fondo los conceptos involucrados en la administración de reglas de negocio, se analizan tecnologías y estándares existentes y estudia el mercado actual.

La fase 3 resume y documenta un estado del arte preliminar sobre el área de estudio.

En la fase 4 se revisan las capacidades de GeneXus con respecto a la administración de reglas de negocio y compara con el enfoque BRA (Business Rules Approach).

En la fase 5 se realiza una propuesta para un manejo dinámico de las reglas de negocio sobre la plataforma GeneXus.

La fase 6 experimenta con motores de reglas de negocio en diferentes plataformas; se decide sobre cuál motor se va a trabajar y qué plataforma se va a utilizar.

En la fase 7 se diseña una solución que satisfaga la implementación de un escenario de uso básico integrado al IDE (Integrated Development Environment) GeneXus.

La fase 8 generaliza un prototipo agregando nuevos casos de uso.

La fase 9 analiza posibilidades para que el experto del dominio pueda entender e influir sobre las reglas de negocio estudiando posibles enfoques para la especificación de reglas de negocio.

En la fase 10 se realiza un prototipo sobre el lenguaje de reglas diseñado y modifican los casos de uso para utilizar éste enfoque. Se implementa un editor externo de reglas para que el experto de negocio tenga la posibilidad de realizar cambios en el ambiente de producción donde ejecuta la aplicación.

La fase 11 comprende la integración de todos los artefactos desarrollados en un único prototipo dentro de lo que es el ecosistema GeneXus.

La fase 12 se ejecuta durante todo el proyecto ya que implica la recolección y publicación de información en un Wiki[1] para su posterior procesamiento.

Finalmente, en la fase 13 se prepara el informe final y la presentación del mismo.

1.5. Organización del documento

El informe se encuentra organizado en seis capítulos y siete anexos. El resto del documento se organiza de la siguiente forma.

El capítulo 2 estudia el estado del arte, así como los conceptos más importantes y los fundamentos para el correcto entendimiento del trabajo. Se presentan los principales productos disponibles en el mercado, estándares existentes y trabajos académicos relacionados.

En el capítulo 3 se expone la propuesta a nivel de la plataforma GeneXus para el tratamiento de reglas dinámicas. Esto incluye la definición de reglas en sí y cómo se realiza su procesamiento; el ciclo de vida de los conceptos que sustentan a las reglas de negocio y cómo se gestiona la evolución de las reglas cuando la aplicación se encuentra en producción.

El capítulo 4 presenta un prototipo desarrollado para evaluar la propuesta del punto anterior. Se presenta la arquitectura del sistema desarrollado, los nuevos artefactos incluidos en la plataforma GeneXus para incorporar capacidades de administración de reglas dinámicas, la implementación en las diferentes plataformas de ejecución, y su posterior administración en el ambiente de ejecución.

En el capítulo 5 se resumen los resultados obtenidos al evaluar el prototipo utilizando los casos de uso más importantes.

Finalmente, en el capítulo 6 se exponen las conclusiones, trabajos a futuro y reflexiones finales.

Relacionado con la sección de apéndices, en el apéndice A se detalla el *Manifesto de reglas de negocio* donde se sintetiza los principios fundamentales del enfoque.

En el apéndice B se estudian las principales empresas y herramientas relacionadas al área de estudio.

El apéndice C resume principales estándares y lenguajes de definición de reglas de negocio tanto a nivel académico como desarrollados realizados por la industria.

El apéndice D expone los algoritmos más importantes que sustentan el enfoque de administración de reglas de negocio.

En el apéndice E se estudian las principales herramientas del mercado (preferentemente de código abierto) en el contexto de las plataformas de ejecución GeneXus.

[1] `http://c2.com/cgi/wiki`

El apéndice F define un lenguaje para representar reglas de negocio teniendo en cuenta dos niveles de abstracción: el primero de "alto nivel" orientado al experto del negocio y el segundo un lenguaje técnico orientado al desarrollador GeneXus.

En el apéndice G se resumen los casos de uso desarrollados utilizando el prototipo construido.

2. Estado Del Arte

En este capítulo se describen diferentes aspectos relacionados con el estado del arte.

A continuación se presenta una breve reseña sobre el uso de Sistemas Expertos y cómo han evolucionado a lo largo de la historia hasta llegar al uso actual como mecanismo para lograr altos niveles de parametrización de aplicaciones. Se introducen conceptos básicos necesarios para comprender el trabajo y posteriormente se formaliza el enfoque de reglas de negocio detallando arquitectura y componentes involucrados.

Se realiza una revisión del estado del arte incluyendo un análisis de las herramientas más importantes, estándares más utilizados y un estudio de mercado y tendencias relacionado al enfoque de reglas de negocio. Además, se estudia la base algorítmica que sustenta el enfoque resumiendo diferentes trabajos académicos relacionados al área de estudio.

2.1. Introducción

La Administración de reglas de negocio surge como una disciplina de las áreas de Inteligencia Artificial [1] y Sistemas Expertos [209]; forma parte de una clase de sistemas inteligentes a los que se les denominan *sistemas basados en reglas*. Otro tipo de sistemas inteligentes incluye a las redes neuronales, lógica difusa, algoritmos genéticos etc.

2.1.1. Los Comienzos

Los sistemas expertos parten de la Inteligencia Artificial en la década del sesenta. Alan Newell y Herbert Simon [187] desarrollaron un programa denominado GPS (General Problem Solver; solucionador general de problemas) que podía trabajar con criptoaritmética[1], torres de Hanoi [225] y otros problemas similares. Lo que no podía hacer el programa GPS era resolver problemas del mundo real, tales como un diagnóstico médico.

Algunos investigadores cambian el enfoque del problema restringiendo el análisis a un dominio específico e intentando simular el razonamiento de un experto humano.

[1] ciencia y arte de resolver criptogramas, los cuales son rompecabezas aritméticos cuyo objetivo es determinar una sustitución de números por letras, de manera que la suma resultante sea correcta aritméticamente.

A mediados de la década del sesenta, un equipo dirigido por Edward Feigenbaum comenzó a desarrollar sistemas expertos utilizando bases de conocimiento definidas minuciosamente. En 1967 se construye DENDRAL [146], que se considera el primer sistema experto. Se utilizaba para identificar estructuras químicas moleculares a partir de su análisis espectrográfico[2].

También en el contexto de la Medicina existen muchísimos sistemas expertos que usan tecnologías de reglas para ayudar en lo que es el diagnóstico de enfermedades y en la medida de lo posible sugerir tratamientos relacionados.

El proyecto MYCIN [132] (iniciado en la Universidad de Standford en la década del setenta) es el caso más conocido. El objetivo del sistema MYCIN fue aconsejar a los médicos en la investigación y determinación de diagnósticos en el campo de las enfermedades sanguíneas. El sistema MYCIN al ser consultado por el médico, solicita primero datos generales sobre el paciente: nombre, edad, síntomas, etc. Una vez conocida esta información, el Sistema Experto permite identificar posibles infecciones bacterianas en la sangre y recomendar una terapia adecuada para dicha infección. Además, este sistema tenía la característica de razonar hacia atrás, es decir, indicar los motivos por los cuáles toma una determinada decisión.

Las investigaciones realizadas por la *Stanford Medical School*[3], develaron que MYCIN tuvo una tasa de aciertos de aproximadamente el 65 %, lo cual mejoraba las estadísticas de la mayoría de los médicos no especializados en el diagnóstico de infecciones bacterianas (el dominio en el que MYCIN estaba especializado). Los médicos que trabajaban exclusivamente en este campo conseguían una tasa de acierto del 80 %.

A partir del éxito de MYCIN; se extiende el uso de este tipo de sistemas en el diagnóstico de enfermedades en otras aplicaciones médicas como es el apoyo en intervenciones quirúrgicas y sistemas de aprendizaje para el entrenamiento de estudiantes y/o médicos.

A inicios de la década del 80 se produce la revolución de los Sistemas Expertos. El primer programa que logra salir del laboratorio se trata de un sistema desarrollado para la DEC (Digital Equipament Corporation) denominado XCON [123] que tenía el objetivo de configurar todos los computadores que comercializara la DEC. El proyecto presentó resultados positivos y se empezó a trabajar en el proyecto más en serio a finales de 1978. El XCON supuso un ahorro de cuarenta millones de dólares al año para la DEC. Sin embargo en el 87; XCON empieza a no ser rentable ya que se llegaron a gastar más de dos millones de dólares al año en tareas de mantenimiento.

[2] Un espectrograma representa la energía del contenido frecuencial de una señal según ésta varía a lo largo del tiempo

[3] `http://med.stanford.edu/`

2.1.2. Sistema Experto

El término *Sistema Experto* (Expert System en inglés) [163] es utilizado en el desarrollo de Software como mecanismo para recomendar una solución a un problema, generalmente a partir de heurísticas sobre la información disponible.

Típicamente, los Sistemas Expertos se construyen en torno a un Motor de Inferencia que aplica mecanismos como la deducción, abducción e inducción (detallado en la sección 2.8) a partir de un conjunto de reglas de inferencia para llegar a una solución.

A partir de los 90, con el desarrollo de la informática y la aparición de los microprocesadores (Apple y compatibles IBM); se produce un amplio desarrollo en el campo de la Inteligencia Artificial y en particular en los sistemas expertos.

A finales de 1990 surgen los primeros sistemas inteligentes denominados motores de evaluación de reglas, aportando su valor al externalizar de las aplicaciones cierta lógica del negocio a través de términos, hechos y reglas [119].

Un motor de evaluación de reglas es un tipo específico de Sistema Experto en donde el conocimiento se representa de la forma de reglas, generalmente como sentencias (*If <condición> then <sentencias>*).

2.1.3. Actualidad

A mediados de la década del 2000, surge la necesidad de automatizar y administrar las reglas de negocio y decisiones; aparecen los Gerenciadores de reglas de negocio [221] y administradores de decisiones corporativos. Se busca hacer explícitas las restricciones sobre los datos, esto es las reglas del negocio, desligándolas lo máximo posible de las aplicaciones y por consiguiente de los sistemas de información.

La tecnología de reglas de negocio a través de los motores de evaluación de reglas y gerenciadores de reglas empezó a utilizarse en marcos donde cada vez más se necesitan sistemas flexibles, como por ejemplo en:

- *Servicios Financieros*: En base al historial del cliente, factores externos y restricciones representadas como reglas de negocio; se calcula la factibilidad de que el cliente obtenga un préstamo.
- *Seguros*: Las empresas de seguros generalmente tiene mucho de su conocimiento de administración, proyección de pólizas y seguros en tecnología de reglas de negocio. Cuando se analiza un seguro o póliza para un posible cliente, se toman en cuenta diferentes características del cliente obteniendo un puntaje (comúnmente denominado *scoring*) que define la viabilidad de obtener el beneficio correspondiente.

Un uso interesante de la tecnología de reglas de negocio en el contexto de la reingeniería de software se presenta en los entornos legados, como es el caso del sistema Mainframe.

La mayoría de las compañías *Fortune 500*[4] – incluidas las empresas de los sectores de Seguros, Banca, Administración Pública y Telecomunicaciones - cuentan con aplicaciones de misión crítica explotadas en entornos Mainframe. En muchos casos, estas aplicaciones están escritas en lenguajes como *COBOL*, lo que requiere para su mantenimiento perfiles de programador muy especializados y cada vez más difíciles de encontrar en el mercado laboral. Como resultado, las políticas de negocio escritas en código *COBOL* no pueden ser revisadas o modificadas de una forma sencilla, lo cual ralentiza los tiempos de respuesta al negocio.

Existe tecnología que utiliza reglas de negocio especializada en este tipo de escenario con el objetivo de simplificar los esfuerzos de modernización de sistemas legados. Productos específicos para este nicho de mercado son:

- *IBM ILOG* con ILOG rules for COBOL[5].
- *FICO* con Blaze Advisor[6].

Esto significa que las reglas de negocio pueden generar un puente que garantice la continuidad de las políticas de negocio desde los sistemas legados a una arquitectura más moderna como puede ser la orientación a servicios y que garantice su aplicación consistente en todos los entornos.

Otro escenario en donde se plantea el uso de reglas de negocio es el mantenimiento de los sistemas. Según estudios realizados por Tony Morgan en [184], detalla que el 75 % del presupuesto de Tecnologías de la Información se gasta en el mantenimiento de los sistemas.

El esquema de reglas de negocio no reducirá mágicamente la necesidad de mantenimiento a cero de los sistemas, pero aislando puntos de flexibilidad que provea una buena trazabilidad entre la definición de la lógica de negocio y su implementación puede reducir significativamente costos asociados al mantenimiento de software.

2.2. Desafíos del Enfoque BRA

La especificación de las reglas de negocio generalmente se encuentra contenida en las aplicaciones y por lo tanto está estrechamente ligada con factores técnicos como los lenguajes de programación, manejadores de bases de datos, etc.

Un cambio en la realidad del negocio que genere un cambio en las reglas empresariales implica modificar las aplicaciones que contienen dichas reglas, lo que trae aparejado una baja productividad.

[4] `http://money.cnn.com/magazines/fortune/fortune500/`
[5] `http://www-142.ibm.com/software/products/es/es/websilogruleforcobo/`
[6] `http://www.fico.com/en/Company/News/Pages/11-08-2005.aspx`

Algunos de los desafíos que enfrenta el enfoque de reglas de negocio (BRA - Business Rules Approach) [204, 222] versus los métodos convencionales son:

- Las reglas de negocio le pertenecen al negocio y por lo tanto el departamento de sistemas generalmente no las comprende totalmente.
- Las reglas de negocio pueden ser muy volátiles y cambiar a menudo siendo necesario una administración adecuada de este tipo de restricciones.

Debido a las características antes mencionadas, los métodos convencionales de automatización de reglas de negocio pueden forzar a las organizaciones a algunos de los siguientes desafíos:

- Falta de visibilidad en cómo las aplicaciones aseguran el cumplimiento de las reglas de negocio. Las aplicaciones se convierten en grandes "cajas negras" y las reglas de negocio críticas quedan "lockeadas" en las líneas de código de la aplicación, en la base de datos o en otras ubicaciones.
- Mantenimiento de Software. Muchas políticas de negocio son volátiles y tener la posibilidad de cambiarlas rápidamente se convierte en un reto. Cuando las reglas de negocio se encuentran codificadas en la aplicación, genera ciclos de test más largos, aumentando el retrabajo y los costos de corrección.
- Conocimiento diseminado. Cuando las reglas de negocio se implementan utilizando los métodos convencionales, tienden a diseminarse a lo largo del código, bases de datos, manuales, planillas Excel, etc. Ante un cambio de requerimiento, puede ocurrir que sea necesario realizar modificaciones en varios componentes y el personal técnico descubra que no conoce con suficiente detalle cómo funcionan las aplicaciones. Esto implica que se generen reglas de negocio ambiguas, incluso conflictivas, dependiendo de cómo fueron implementadas.

2.3. Fundamentos

El *Business Rule Group* [100] surge a inicios del 2000 con el propósito de proveer una visión global del enfoque de reglas en el desarrollo de Software. Sintetiza los objetivos fundamentales y establece los fundamentos para que las reglas de negocio cumplan un rol principal en la especificación de requerimientos considerando a:

- Los requerimientos como elementos principales, nunca como secundarios administrados en forma declarativa.
- Las reglas como elementos *independientes* de los procesos.
- Las reglas al servicio del negocio y no de la tecnología.

Por mas información consultar el Anexo A.

2.4. Uso de Reglas de Negocio

El enfoque de reglas de negocio es una metodología y seguramente un conjunto de tecnologías en la cual se captura, publica, automatiza y modifican las reglas desde un punto de vista estratégico. El resultado es un sistema de reglas de negocio en el cual las reglas se encuentran separadas lógica y hasta físicamente de otros aspectos del sistema y son compartidas entre diferentes aplicaciones.

Algunos puntos a considerar para determinar si debería o no utilizar un motor de ejecución de reglas [121] son:

- Complejidad de la aplicación.
- Ciclo de vida de la aplicación.
- Variabilidad de la la aplicación.

La primera diferencia que presenta un motor de evaluación de reglas de negocio es cómo se expresan las reglas. En vez de embeberlas en el código fuente del programa se codifican en un lenguaje de reglas de negocio que varía dependiendo del motor utilizado.

Encapsular decisiones basadas en reglas, entre otras cosas genera las siguientes ventajas:

- *Mayor modularidad.* Separar las reglas de negocio permite su reutilización desde cualquier aplicación. Se aseguran servicios modulares y reutilizables al utilizar un motor de evaluación de reglas. Dado que las reglas y por lo tanto la lógica del negocio se encuentra aislada, las responsabilidades en el desarrollo e integración se encuentran claramente definidas.
- *Mayor consistencia.* Todas las reglas que se relacionan en forma lógica pueden organizarse y mantenerse en una ubicación centralizada, minimizando la duplicación de código y aumentando la consistencia.
- *Simplicidad.* Se captura la regla de negocio en un formato que el analista puede entender. Se pueden definir diferentes formas de representación de reglas, como por ejemplo árboles y tablas de decisión.
- *Auto-descriptiva y Entendible.* Una tabla de precios en una base de datos nunca podrá comunicar el escenario completo tan bien como una tabla de decisión. Se aumenta la colaboración entre las áreas de negocio y el departamento de sistemas al lograr un lenguaje común de comunicación.
- *Test independiente.* Dado que las reglas se testean en forma separada, además de asegurar la consistencia minimiza el esfuerzo a realizar cuando se pone en producción la aplicación.

Uno de los retos más importantes en el uso del enfoque de reglas es encontrar aquellas políticas empresariales y restricciones que rigen el negocio [158].

2.5. Definición de Regla de Negocio

No existe una definición única para el término regla de negocio (Business Rule).

El *Business Rule Group* [100] la define como:

> *"Una declaración que define o restringe algún aspecto del negocio. Su cometido es declarar una estructura relevante para el negocio o controlar e influir sobre el comportamiento del mismo"*

Gartner [24] las define como:

> *"Políticas de negocio implícitas y explícitas que definen y describen una acción de negocio, donde las reglas implícitas son aquellas que se encuentran embebidas dentro de las aplicaciones"*

Hay & Healey [159, 160] las define como:

> *"una sentencia que define o restringe algún aspecto del negocio"*

De manera informal, se pueden concebir las reglas de negocio como un conjunto de condiciones que expresan cómo deben llevarse a cabo los procesos del negocio, de forma tal que su resultado sea aceptable. Describen o representan restricciones sobre el comportamiento del negocio [206].

En términos generales, las reglas de negocio pueden describirse como instrucciones que indican lo siguiente:

> *"emprenda una acción o un conjunto de acciones, si hay una condición o combinación de condiciones o si se produce un suceso o una combinación de sucesos"*

Representan un conjunto de prácticas de negocio estándar que se deben de aplicar en forma consistente dentro de las actividades del negocio. Son el componente más dinámico de cualquier aplicación. Por lo tanto, su identificación y externalización mejora la capacidad de adaptación de las organizaciones a cambios en la industria y competencia.

Las reglas de negocio deben ser expresadas declarativamente, no en forma procedural; deben indicar lo *qué* se quiere asegurar sin indicar *cómo*.

Es posible declararlas en lenguajes formales, pero también deben ser expresadas en "*lenguaje natural*" [186], facilitando su compresión para todos los involucrados en el negocio.

Estas *reglas* siempre deben de estar basadas en *términos* y *hechos* (ver Figura 2.1) [183]; no es correcto definir reglas sobre términos que no han sido definidos previamente.

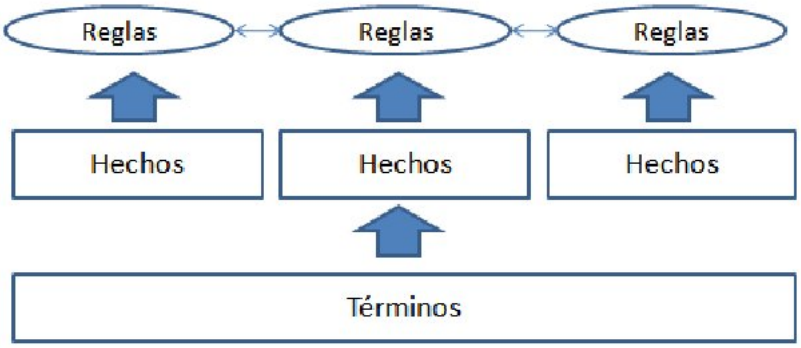


Figura 2.1.: Términos, Hechos y Reglas

2.5.1. Término

Los términos son palabras o frases en "*lenguaje natural*" que los involucrados en el negocio reconocen y comparten.

Un término posee un significado para el negocio que debe ser comprendido y compartido.

Debe poseer una definición precisa que se formaliza en un Glosario de términos, Vocabulario u Ontología [157] que describa el dominio del problema.

Puede definir entre otras cosas:

- Un concepto, por ejemplo "*un Cliente*".
- Una propiedad de un concepto, por ejemplo "*Crédito máximo de un Cliente*".
- Un valor, como por ejemplo "*Femenino*".
- Un conjunto de valores, como por ejemplo los días laborales; "*lunes a viernes*".

Algunas características de los términos son:

- un término es *no ambiguo*; esto significa que dado el contexto del negocio, un término está representado una sola vez y no se solapa con otro término.
- es *básico* y *atómico*; básico implica que no puede ser derivado ni calculado a partir de otros términos; atómico significa que es indivisible.
- es *cognoscible*, por lo que representa "*algo*" que conocemos.
- debe estar *definido*; el concepto que un término representa no debe darse por sentado, debe ser definido en forma explícita.
- un término se representa mediante un *sustantivo* o nombre en singular.

2.5.2. Hecho

Un *hecho* es una expresión que representa la conexión lógica entre dos términos. Algunas características de los hechos son:

- *No es ambiguo*; dado el contexto del negocio un hecho está representado una sola vez y no se solapa con otro hecho.
- *Se basa en términos*; está construido a partir de términos y tiene la siguiente estructura: <*Término*> <*Relación*> <*Término*>, donde <*Término*> indica la ocurrencia de un término del catálogo de conceptos y <*Relación*> indica un verbo que relaciona ambos términos. De lo anterior se infiere que en un hecho aparecen dos y sólo dos términos conectados por un verbo.
- Los *términos* y *hechos* van a definir la semántica de las reglas.

Algunos ejemplos de hechos.

- Un Cliente puede realizar órdenes de compra.
- Una Orden está compuesta por varios ítems.
- Los ítems de una orden detallan un único producto.

2.5.3. Evento

De manera formal, un evento puede ser definido como cualquier cambio de estado.

Informalmente, es una instancia que sucede, y que requiere la respuesta del negocio. Por ejemplo, la generación de una orden de compra es un evento válido en una aplicación de facturación. Este evento requiere una respuesta por parte del sistema en su conjunto. En el contexto asociado a los motores de ejecución de reglas, el ejecutor de reglas deberá determinar la respuesta del sistema al evento tomando en cuenta las reglas involucradas.

2.5.4. Regla

Una regla es una clase de instrucción o comando que se aplica en una situación determinada; generalmente se pueden escribir con sentencias de la forma "*If-Then*".

A la sección "*If*" se le denomina parte izquierda (*LHS* - Left-Hand Side), predicado o premisa, que consta de una serie de patrones que especifican los hechos (o datos) que causan que la regla sea aplicable.

A la sección "*then*" se le denomina parte derecha (*RHS* - Right-Hand Side), acciones o conclusiones, que detalla las acciones que se deben realizar en caso de que se satisfagan las premisas (sección "*If*").

Por ejemplo, supongamos que dado un *Vehículo* y *Conductor*; si las siguientes condiciones se satisfacen, entonces se debe aumentar un 20 % el seguro del vehículo. La regla se puede especificar de la siguiente manera:

```
If: (premisa)
   el vehículo es rojo
   el vehículo es de tipo deportivo
   el conductor es masculino
   el conductor tiene de 16 a 25 años.
Then: (consecuencia)
   aumentar un 20% el seguro Premium
```

Una regla de negocio entonces, es una sentencia declarativa que aplica lógica o un cálculo predeterminado. El resultado de una regla es el descubrimiento de nueva información o la toma de una acción a partir de una decisión.

Algunos ejemplos:

- El total de una orden de compra es la suma de las líneas de la misma.
- Un cliente nuevo no puede realizar una orden de compra que exceda los $1000.
- El envío de órdenes a clientes que no hayan pagado la última factura, serán retenidos hasta que se realice el pago.

En definitiva, el manejo de reglas de negocio según el enfoque BRA [204] implica:

- Extraer las reglas de negocio de procesos y procedimientos.
- Expresarlas en forma declarativa utilizando términos y hechos.
- Ejecutar un motor de evaluación de reglas con capacidades de inferencia\razonamiento y permitir que los analistas de negocio modifiquen las reglas.

2.5.5. Analogía Reglas de Negocio versus Cuerpo Humano

Una forma interesante de conceptualizar las reglas de negocio se detalla en [205]; utilizando una analogía con el sistema mecánico del cuerpo humano se presentan los componentes principales relacionados con las reglas de negocio. Se divide al cuerpo humano en los siguientes componentes:

- *Estructura* provista por los huesos, organizados e interconectados para formar el esqueleto.
- *Potencia* provista por los músculos, los cuales se conectan con los huesos, permitiendo que el cuerpo tenga la capacidad de movimiento.

- *Control* provisto por el sistema nervioso, el cuál conecta a los músculos en forma indirecta a través del cerebro. Las respuestas a cualquier impulso son coordinados a través del disparo de impulsos nerviosos.

Los tres componentes son esenciales y están interconectados. A su vez cada componente se encuentra especializado en una tarea, rol o responsabilidad. Relacionado con el enfoque de administración de reglas [204]:

- Los *términos* y *hechos* forman la estructura, por lo que se pueden ver como si fuera el esqueleto del cuerpo humano.
- Los músculos constituyen los *procesos*, detallan cómo se logran los objetivos, operando con los *términos* y *hechos* del punto anterior.
- El sistema nervioso se corresponde con las estrategias, la memoria y los mecanismos para procesar los *eventos* que suceden.

Las *reglas* se asemejan a los nervios, ya que conectan los diferentes *procesos*. Utilizando *reglas* a partir de los *términos* y *hechos* definidos, se controlan los *procesos* de negocio.

2.5.6. Clasificación de alto nivel de Reglas

Una clasificación de alto nivel de las reglas de negocio [203] se puede dividir en tres grandes grupos mutuamente excluyentes, detallados en la Figura 2.2.

- Restricción.
- Producción.
- Proyector o Inferencia.

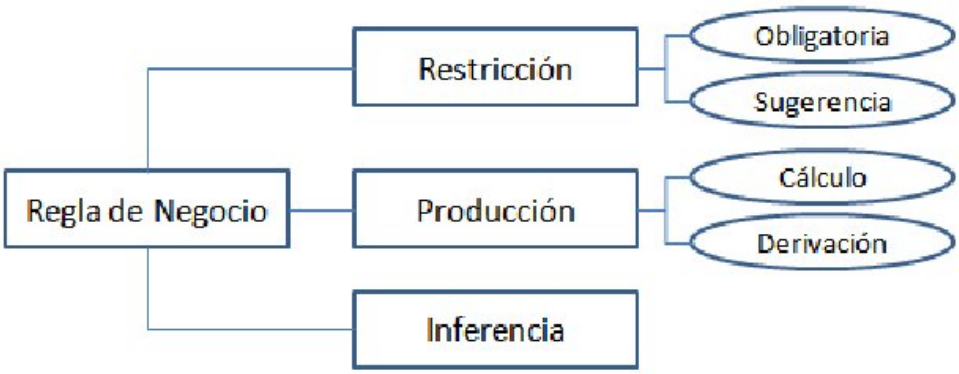


Figura 2.2.: Clasificación de Reglas

Regla de Restricción

Las reglas de negocio de tipo Restricción son una sentencia lógica que expresa una afirmación que debe ser válida en todo estado o transición entre estados.

Su fin es rechazar un cambio de estado en caso que se produzca un estado inválido para asegurar la consistencia del sistema, favoreciendo una alta calidad de datos.

Si una regla es catalogada como una restricción, el motor de evaluación de reglas de negocio debe considerarla en todos los eventos que pueda producir un cambio de estado que viole la restricción. Algunos ejemplos:

> *Los clientes no pueden retirar dinero si sus cuentas están sobregiradas.*
> *Una fecha es obligatoria por cada transacción.*
> *Los descuentos en la mercadería solo se pueden aprobar por administradores.*

Regla de Producción

Las reglas de negocio de tipo Producción calculan un valor o derivan un resultado, produciendo un cálculo en forma automática. Las mismas aseguran que el cálculo siempre es actual y correcto, distinguiéndose dos subcategorías:

- *Regla de cálculo*: calcula un valor por medio de una fórmula que contiene operadores estándar (suma, resta, división, multiplicación, promedio, etc.).
- *Regla de derivación*: el resultado de la evaluación es un valor lógico (verdadero, falso) y están basados en operaciones lógicas.

Regla de Cálculo Las reglas de Cálculo definen un cómputo. A su vez, un cómputo es una sentencia completa que provee un algoritmo para llegar a un valor, pudiendo incluir operaciones como suma, diferencia, producto, cociente, cantidad, máximo, mínimo, promedio. Por ejemplo:

> *El total de una factura se calcula como la suma de las líneas de factura más la tasa correspondiente.*

Regla de Derivación Una regla de Derivación infiere un valor lógico (verdadero o falso), basado en operaciones lógicas (*AND*, *OR*, *NOT*, etc).

Ejemplo:

> *Un proyecto se encuentra en riesgo cuando se supera el presupuesto asignado al mismo.*

Regla de Inferencia

Las reglas de negocio de tipo Inferencia son reglas que infieren nuevo conocimiento\ejecutan procesos o procedimientos automáticamente.
Se distinguen las siguientes subcategorías:

- *Proyectores de inferencia*: son reglas que infieren que algo es verdadero bajo ciertas circunstancias, si dichas circunstancias no se cumplen no se afirma que sea verdadero ni falso.
- *Conmutadores de reglas*: habilitan o deshabilitan reglas bajo ciertas circunstancias; también se les conoce como "*regla toogle*".
- *Conmutadores de procesos*: son análogos a los anteriores para los procesos.
- *Conmutadores de datos*: son proyectores que modifican registros en la base de datos bajo ciertas circunstancias.
- *Disparadores de reglas*: causan la ejecución de la regla especificada bajo ciertas condiciones.
- *Disparadores de procesos*: son análogos a los anteriores para los procesos.

Se detalla a continuación un ejemplo de regla de proyección:

> *Si un cliente tiene el status preferido, sus órdenes de compra califican por un descuento del 20 %.*

2.6. Formas de implementar Reglas de negocio

Las reglas de negocio se pueden implementar de diferentes maneras. Algunos ejemplos en los cuales varía el nivel de sofisticación de la solución planteada son:

- *Código*: implementar las reglas de negocio en un lenguaje de programación de propósito general.
- *Scripts y lenguajes dinámicos*: se separan las reglas de negocio del código procedural, por lo que la actualización de alguna regla implica actualizar uno o varios scripts o modificar el código del lenguaje dinámico que se debe evaluar.
- *Restricciones en la base de datos*: uso del mecanismo para definir restricciones sobre la base de datos, desde restricciones sobre valores para asegurarse que se cumplan las restricciones sobre un dominio (un cliente debe tener al menos 18 años) hasta restricciones sobre estructura de datos para restringir las relaciones entre entidades, como es el caso de la integridad referencial.

- *Procedimientos almacenados*: implementación de las reglas de negocio utilizando procedimientos almacenados. En este caso se decide mantener las decisiones cerca de los datos.
- *Triggers*: mecanismo similar a los procedimientos almacenados, con la diferencia que los triggers ocurren en forma automática cuando sucede el evento sobre el cual se declaró el trigger.
- *Componente de reglas de negocio*: utilizar un lenguaje declarativo para la implementación de las reglas de negocio, como por ejemplo *Prolog* [216], *Clips* [11], *Jess* [44].
- *Motor de evaluación de Reglas de negocio*: uso de un motor de evaluación de reglas de negocio.
- *Gerenciador de Reglas de negocio*: dejar externo a la aplicación las reglas de negocio que tienen un impacto importante en la toma de decisiones y sean factibles de cambiar frecuentemente usando un motor de evaluación de reglas para ejecutarlas.

En el contexto del trabajo interesa investigar componentes de evaluación de reglas de negocio, donde las reglas se administren en forma independiente a las aplicaciones y procesos del sistema.

2.7. Arquitectura

Desde una perspectiva de Sistemas de Información, las reglas de negocio cambian más frecuentemente que el código de una aplicación. Los motores de evaluación de reglas (*Rules Engines* o *Inference Engines*) [135] son componentes de software que separan las reglas de negocio del código de aplicación, permitiendo que los usuarios modifiquen las mismas frecuentemente sin la necesidad de intervención de una persona con conocimiento técnico, de manera de permitir que las aplicaciones sean más flexibles y adaptables.

En términos generales, consta de los componentes que detalla la Figura 2.3.

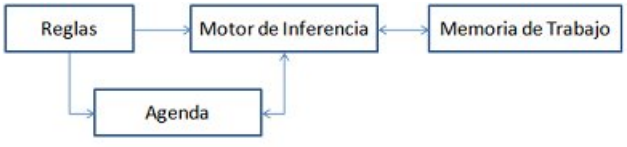


Figura 2.3.: Componentes de un motor de evaluación de reglas

El motor de evaluación de reglas provee los mecanismos necesarios para ejecutar reglas (también denominadas *producciones*) y alcanzar algún objetivo; las reglas

constan de una precondición y una acción y utilizan lógica de primer orden para representar el conocimiento.

2.7.1. Base de Reglas

La Base de Reglas (*RuleBase*, *Production Rules*) contiene el conjunto de reglas por el cual se rige el negocio. Es decir, contiene todas las reglas que el sistema conoce. No existen reglas que influyan en el comportamiento del negocio por fuera de este conjunto.

Existen diferentes maneras de organizar las reglas dentro de la base de reglas para su gestión. Entre ellas se pueden destacar:

- Área(s) del negocio que aplica.
- El propósito que persigue.
- Metadata administrativa como fecha de creación, modificación, intervalo de tiempo en el cual se debe aplicar, etc.
- Términos que la componen.

De igual forma, en la base de reglas es posible almacenar información de relaciones entre reglas. Entre las relaciones más significativas se encuentran:

- *Excepción*: Una regla es una excepción de otra.
- *Conmutación*: Una regla habilita o inhabilita a otra.
- *Conflicto*: Una regla entra en conflicto con otra; es decir, una regla no puede ser satisfecha si genera un estado invalido en una segunda regla.

La Base de Reglas generalmente se almacena en un Repositorio que será accedido por las aplicaciones cuando sea necesario evaluar las reglas en un contexto de ejecución.

2.7.2. Agenda

Típicamente, un evaluador de reglas contiene cientos o miles de reglas y es probable que en un momento dado se activen varias reglas. La lista de reglas que son potencialmente ejecutables se almacenan en lo que se denomina *Agenda*.

Al conjunto de reglas activadas que tienen posibilidad de ser ejecutadas se les denomina *grupo conflictivo de reglas* (*conflict set*). Al proceso de ordenar las reglas para ser disparadas se le denomina resolución de conflictos; el resultado de la resolución de conflictos es una lista ordenada de activaciones de reglas que es la *Agenda* propiamente dicha.

2.7.3. Memoria de Trabajo

La memoria de trabajo (*working memory* o *fact base*), contiene toda la información (conocimiento) con la cuál un motor de inferencia trabaja. Por lo general se almacenan tanto las premisas como las conclusiones de las reglas.

El diseñador de reglas se encarga de decidir qué información se almacena en la memoria de trabajo y en forma explícita deberá cargar en el contexto de ejecución los hechos relevantes sobre los cuáles se aplicará la base de reglas existente.

Un *hecho* es la unidad de información más pequeña con la que se puede trabajar en la memoria de trabajo. Generalmente se ofrece un conjunto de operaciones para interactuar con la memoria de trabajo como por ejemplo:

- *Assert* o *Insert*: agrega hechos a la memoria de trabajo (*working memory*).
- *Retract*: elimina un hecho de la memoria de trabajo.
- *Modify*: modifica un hecho al cambiar ciertas condiciones.
- *Clear*: elimina todos los elementos.
- *Deffacts*: define el contenido inicial de la memoria de trabajo.
- *Facts*: despliega el contenido de la memoria de trabajo.
- *Reset*: inicializa la memoria de trabajo.
- *Watch*: imprime diagnóstico cuando se dan modificaciones sobre los hechos como por ejemplo cuando hay cambios en la memoria de trabajo.

2.7.4. Comparación de Patrones

El motor de inferencia debe decidir qué reglas se disparan y cuándo según el contexto de ejecución, es decir, los hechos que se encuentren definidos en la memoria de trabajo. El propósito del componente que compara patrones (*pattern matcher – matcheo de patrones*) [215] es decir cuándo aplicar cuales reglas según el contenido de la memoria de trabajo.

Generalmente éste es un punto complejo en el diseño de los motores de evaluación de reglas dado que el *pattern matcher* deberá buscar en miles o millones de combinaciones de hechos para encontrar aquellos hechos que satisfagan las reglas.

2.7.5. Reseña Algorítmica

Existen un conjunto de algoritmos que abordan la problemática de "matchear" patrones en forma eficiente. Se detalla a continuación una breve reseña de los más importantes. Consultar el Anexo D por más información.

- *Rete*: es un algoritmo diseñado por Charles L. Forgy [150] para eliminar las ineficiencias del "matcheo" de patrones (pattern matching). El algoritmo construye una red ordenada en forma de árbol que contiene todos los patrones en las condiciones (sólo una vez) y se utiliza para revisar contra la memoria de trabajo evitando ciclos internos guardando los resultados anteriores de las evaluaciones de los hechos en cada iteración. Solo se testean elementos nuevos o modificados en la memoria de trabajo.

- *ReteII*: Algoritmo propietario sucesor de Rete inventado por Charles Forgy que ejecuta entre 100 y 1000 veces más rápido que el original dependiendo de la complejidad de las reglas y objetos que se ingresan a la memoria de trabajo.

- *ReteIII*: Algoritmo propietario que parte del algoritmo ReteII agregando RLT (Relational Logic Technology[7]), fue desarrollado especialmente para hacer posible que los motores de reglas puedan manejar efectivamente un conjunto importante de datos así como también grandes cantidades de reglas.

- *ReteOO* [121]: es una implementación orientada a objetos de Rete, fue adaptado por Bob McWhirter y es utilizado en Drools [13].

- *Rete Plus*: es una implementación propietaria del algoritmo Rete desarrollado por la empresa IBM ILOG [107] agregando algunas extensiones.

- *DRETE*: Distributed Rete [168] es un algoritmo de "matcheo" de patrones desarrollado por Michael A. Kelly y Rudolph E. Seviora. En DRete la fase de "matcheo" de patrones se particiona y paraleliza al nivel de comparación de tokens.

- *TREAT*: Daniel P. Miranker combina el algoritmo Rete con técnicas de optimización derivadas de los motores de bases de datos relacionales. TREAT (TREe Associative Temporal redundancy) [181] no utiliza una red discriminada y no cachea los resultados intermedios en las intersecciones de las redes tal como lo hace Rete; el matching de patrones se recalcula cada vez que se necesita. El algoritmo es una alternativa a Rete cuando el consumo de memoria es un requerimiento a tener en cuenta.

- *LEAPS*: Lazy Evaluation Algorithm for Production Systems [124] es una extensión del algoritmo TREAT. El aporte fundamental de este algoritmo es que las tuplas se evalúan solo cuando se necesitan (lazy evaluation).

- *DETI*: Design-Time Inferencing es un algoritmo de "matcheo" de patrones propietario desarrollado por la empresa Corticon [12].

[7] `http://www.faqs.org/patents/app/20090106063`

- *MatchBox*: es un algoritmo desarrollado por Mark Perlin [202] para determinar la instanciación de tuplas en sistemas basados en reglas que utilizan inferencia hacia adelante.
- *HAL*: Heuristically-Annotated-Linkage [174] presenta una solución diferente para el matcheo de patrones ya que utiliza el conocimiento heurístico embebido en las reglas y los elementos que conforman la base de hechos y los relaciona.

2.7.6. Resolución de Conflictos

Dado que el disparo de las consecuencias de las reglas manipula el conocimiento que existe en un momento dado en el motor de evaluación de reglas de negocio, es importante definir el orden en el cual se dispararán las reglas.

Como se mencionó anteriormente, aplicar un orden a las diferentes activaciones que se produzcan se conoce como *resolución de conflictos*. Algunas estrategias que se pueden utilizar son:

- *Salience*: a cada regla se le puede definir un atributo entero especificando su prioridad; es decir aquellas reglas con los valores más altos tendrán mayor prioridad por lo que se ejecutarán primero.
- *Recency*: se verifica el contador que se asigna a cada hecho en la activación; aquellas activaciones con el mayor contador son los primeros en la agenda.
- *Primary*: se verifica el contador que se asigna para cada hecho en la activación; aquellas activaciones con el menor contador son los primeros en la agenda.
- *FIFO* o *depth*: las reglas activadas recientemente son las que serán disparadas primero.
- *LIFO* o *breath*: las reglas se disparan según el orden de activación, la última activación va a ser disparada en primer lugar.
- *Complexity*: se toma en cuenta la complejidad de las reglas en conflicto; cuanto más compleja la regla (más condiciones tiene), más específica será; las activaciones con la mayor complejidad son las primeras en la agenda.
- *Simplicity*: se toma en cuenta la simplicidad de las reglas en conflicto; cuanto más simple es la regla (menos condiciones tiene) mayor prioridad tendrá, por lo que serán las primeras en la agenda.
- *Orden de carga*: cuando una regla se agrega al conjunto de reglas se le asigna un número de carga único; aquellas reglas con mayor contador serán las primeras en la agenda.
- *Random*: las activaciones se agregan en forma randómica en la agenda.

2.7.7. Motor de Inferencia

El motor de inferencia es uno de los varios mecanismos existentes para aplicar conocimiento sobre los datos. El principal objetivo es aplicar las reglas que se tengan definidas a los datos (hechos) existentes. El primer paso consiste en instanciar un contexto de reglas, este contexto contiene la memoria de trabajo, un conjunto de reglas y una agenda.

Se carga el juego de reglas a partir del repositorio y posteriormente desde la aplicación se insertan los objetos con los cuales va a trabajar el motor de reglas.

Una vez que la aplicación ha insertado los objetos, la aplicación ejecutará un método para disparar todas las reglas (típicamente el método *fireAllRules*).

Es aquí en donde el motor de reglas toma el control y debe determinar qué reglas son factibles de ser ejecutadas según los hechos existentes en la memoria de trabajo.

El Motor de inferencia por lo general realiza las siguientes operaciones:

- Se aplican las reglas a la memoria de trabajo.
- Se comparan los diferentes patrones existentes; se comparan todas las reglas con la memoria de trabajo utilizando el comparador de patrones (*pattern matcher*) para decidir cuáles reglas deben activarse en este ciclo. Como ya se mencionó anteriormente, a ésta lista desordenada de reglas que se activan se le denomina grupo conflictivo de reglas (*conflict set*).
- Se crea una *Agenda* con las activaciones; el grupo conflictivo de reglas es ordenado para formar la agenda con la lista de reglas que serán ejecutadas.
- Se ejecutan las activaciones en un motor de ejecución. La primera regla de la agenda es disparada, por lo que se ejecutan todas las operaciones de la sección de acciones de la misma. Es posible que en las propias acciones de la regla se actualice la memoria de trabajo del motor de ejecución; esto implica que todo el proceso se vuelva a repetir. Es decir, es posible que una regla cambie un objeto y éste cambio implique que nuevas reglas sean factibles de ser ejecutadas, que serán agregadas a la agenda. Este proceso continúa hasta que no hay reglas en la *Agenda*.
- Si bien esta repetición implica una cantidad importante de retrabajo, la mayoría de los motores de evaluación de reglas utilizan técnicas sofisticadas para evitar este retrabajo. En particular, los resultados de comparar patrones en las condiciones de las reglas y la resolución de conflictos para crear la *Agenda* será preservado entre los diferentes ciclos de ejecución, de manera que solo lo esencial será re-ejecutado.
- Se devuelve el control a la aplicación.

A grandes rasgos se ejecuta el ciclo de operaciones que se detalla en la Figura 2.4 en dos fases: determinar reglas que se tienen que disparar y posteriormente la evaluación correspondiente utilizando la agenda.

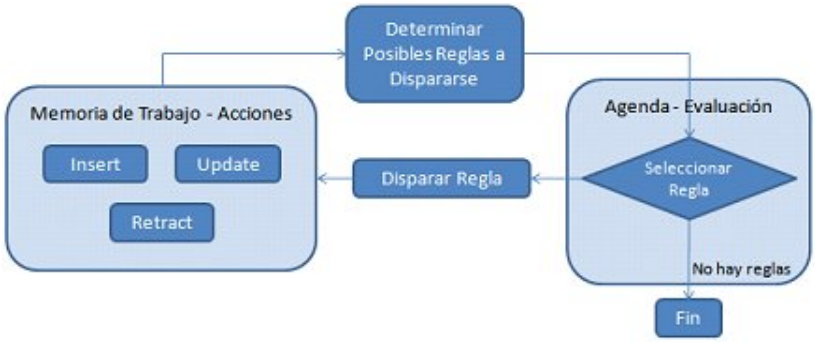


Figura 2.4.: Procesamiento en dos fases

2.7.8. Tipos de Motores

Existen varias clases de motores de evaluación de reglas; destacamos los denominados de *inferencia o encadenamiento hacia delante* (forward chaining) e *inferencia o encadenamiento hacia atrás* (backward chaining).

Encadenamiento hacia adelante

Los Motores de inferencia de tipo *Forward-chaining* [155] comienzan el mecanismo de inferencia con los datos disponibles y utilizan reglas para extraer más datos hasta que se alcanza un objetivo.

Define reglas de la forma "*If <condition> then <action>*", donde:

- Sección *If*: se le considera la *condición* de la regla.
- Sección *Then*: se le considera la parte de *acción* de la regla.

Son útiles para los casos donde los problemas requieren llegar a conclusiones de alto nivel a partir de hechos simples.

Si bien las reglas se definen en forma procedural, las sentencias se ejecutan cuando se satisfacen las condiciones "*If*", y es el motor de ejecución de reglas el cual se encarga de decidir qué reglas disparar y en qué orden. Dado que los datos determinan qué reglas son seleccionadas y posteriormente ejecutadas, a este método se le denomina "*orientado a los datos*".

Cuando el motor de ejecución procesa un hecho en la memoria de trabajo, verifica su base de reglas para ver si alguna de las condiciones es verdadera. En caso afirmativo,

las acciones que detallan las reglas son procesadas, pudiendo cambiar la información de la memoria de trabajo.

Cuando se completa la acción, el motor chequea si la condición de alguna otra regla es verdadera y el proceso se repite hasta que no existan reglas que sean verdaderas.

Un ejemplo de cómo funcionan los motores de inferencia *Forward-chaining* se decriben en la Figura 2.5.

Figura 2.5.: Encadenamiento hacia adelante

Encadenamiento hacia atrás

Los Motores de inferencia de tipo *Backward-chaining* [155] no tienen una analogía con los lenguajes procedurales, si bien también definen hechos utilizando la sintaxis del tipo "*If <condition> then <action>*", se trata de satisfacer las condiciones ejecutando las partes derecha de las reglas para validar si se logran los objetivos que plantea la regla.

Es decir, si la clausula izquierda de una regla es satisfecha parcialmente y el motor puede determinar que al disparar alguna otra regla puede causar que la condición se cumpla en su totalidad, entonces el motor ejecuta la segunda regla, lo que se denomina búsqueda del objetivo o *goal seeking* [155].

Los pasos que ejecuta el algoritmo se describen en la Figura 2.6.

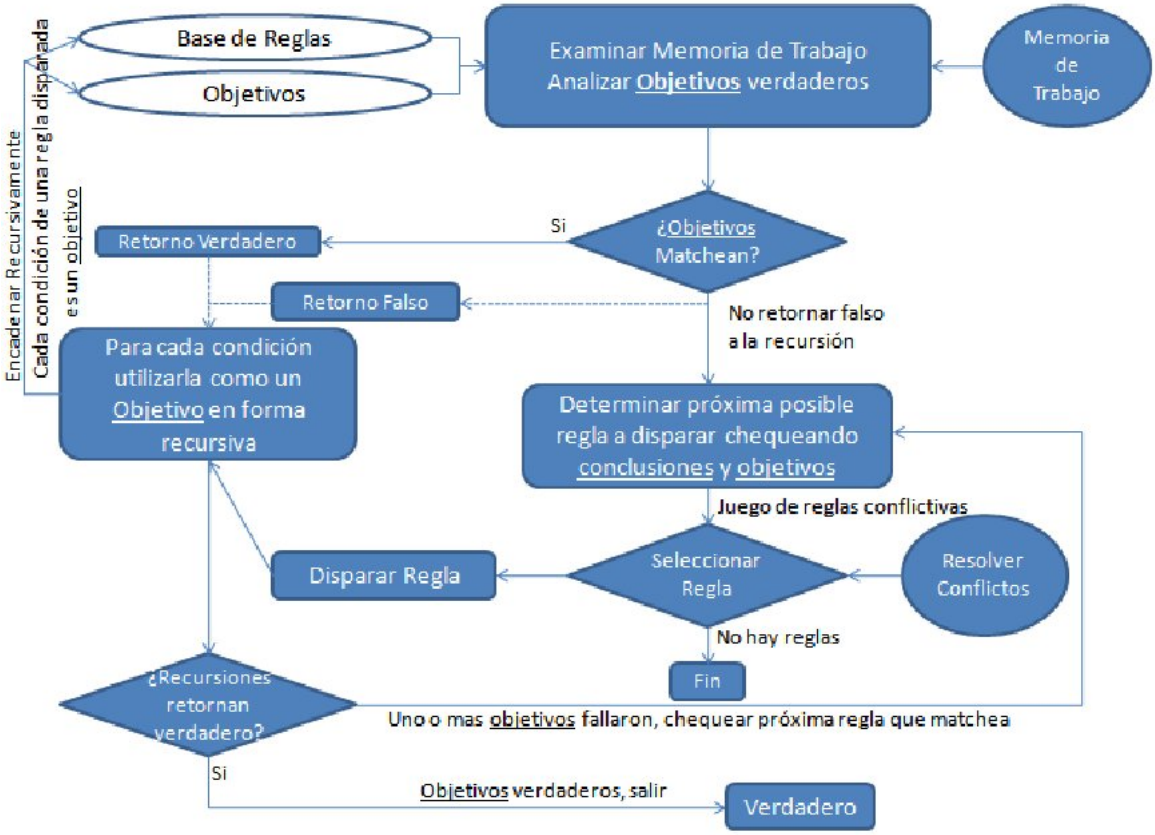


Figura 2.6.: Encadenamiento hacia atrás

El algoritmo realiza las siguientes operaciones:

- Se comienza con una conclusión que el motor trata de satisfacer.
- Si no se puede satisfacer, se buscan conclusiones intermedias que sí se puedan satisfacer (*sub goals*), lo que permite satisfacer alguna parte del objetivo actual (*current goal*).
- Se continúa con este proceso hasta que se prueban las conclusiones iniciales o no hay más sub-conclusiones (*sub goals*) para analizar.

2.8. Base Teórica

Los Motores de reglas de negocio tienen una fuerte base teórica sostenida en lo que es la Lógica [155]. La Lógica es una ciencia formal que tiene el objetivo de estudiar las formas válidas de inferencia. Es el estudio de métodos y principios utilizados para distinguir el razonamiento correcto del incorrecto.

En particular, relacionado con estudio de los motores de evaluación de reglas interesa la Lógica Deductiva, que se interesa en establecer una conclusión a partir de un conjunto de hechos que se sabe son lógicamente correctos. También interesa la Lógica Inductiva, en la que se utilizan hechos particulares para inferir reglas generales. A continuación introduciremos brevemente estos conceptos.

2.8.1. Deducción, Inducción, Abducción

La deducción [224] o método lógico deductivo es un método científico que considera que la conclusión está implícita en las premisas. Es decir que la conclusión no es nueva, se sigue necesariamente de las premisas (`Causa + Regla` $\rightarrow$ `Efecto`). Si un razonamiento deductivo es válido y las premisas son verdaderas, la conclusión sólo puede ser verdadera (se va de lo universal a lo singular).

Ejemplo:

```
Regla: Todas las bolillas de la bolsa x son blancas.
Caso: Estas bolillas provienen de la bolsa x.
Deducción: Estas bolillas son blancas.
```

Los Motores de tipo *Forward-chaining* utilizan la deducción como mecanismo de inferencia.

La Inducción [224] es una modalidad del razonamiento no deductivo que consistente en obtener conclusiones generales a partir de premisas que contienen datos particulares. La inducción permite hipotetizar una regla a partir de un Caso y de una Conclusión (`Causa + Efecto` $\rightarrow$ `Regla`), se va de los singular a lo universal. La inducción no es lógicamente válida sin confirmaciones externas, ya que bastaría una excepción a la regla para que la misma sea falsa.

Ejemplo:

```
Caso: Estas bolillas proceden de la bolsa x.
Caso: Estas bolillas son blancas.
Inducción: En la bolsa x todas las bolillas son blancas.
```

La Abducción [224] es un tipo de razonamiento propuesto inicialmente por Aristóteles en el cual se trabaja con una especie de Silogismo donde la premisa mayor es considerada cierta mientras que la premisa menor es solo probable. Esto significa que la conclusión a la que se puede llegar tiene el mismo grado de probabilidad que la premisa menor. En la abducción, a fin de entender un fenómeno, se introduce una regla que opera en forma de Hipótesis para considerar dentro de tal regla al posible resultado como un caso particular. El razonar en forma abductiva consiste en explicar un fenómeno $\boldsymbol{q}$ mediante $\boldsymbol{p}$ considerando a $\boldsymbol{p}$ como hipótesis explicativa (`Efecto + Regla` $\rightarrow$ `Causa`).

Ejemplo:

```
Regla: Todas las bolillas de la bolsa x son blancas.
Caso: Estas bolillas son blancas.
Abducción: Estas bolillas proceden de la bolsa x.
```

Los Motores de tipo *Backward-chaining* utilizan la abducción como mecanismo de inferencia.

Es importante resaltar que la Inducción y Abducción no son razonamientos válidos sin una ratificación empírica y pese a todas las posibles ratificaciones empíricas, siempre existe el riesgo de una excepción.

2.8.2. Razonamiento Monótono versus No-Monótono

Otro concepto importante que se debe tener en cuenta es el razonamiento monótono versus el razonamiento no-monótono.

La lógica clásica es *monótona* (del inglés Monotonic Logic [155]): cuando se tiene una sentencia A que es una consecuencia lógica de un conjunto de sentencias T, entonces A es también una consecuencia de un superconjunto arbitrario de T. En otras palabras, se establece que las hipótesis de cualquier hecho derivado pueden ser extendidas, es decir, que al agregar nuevo conocimiento no se invalidan las deducciones que ya se realizaron.

El razonamiento utilizando sentido común es diferente, ya que a menudo se obtienen conclusiones plausibles en base a supuestos sobre información incompleta. Puede ocurrir que dichos supuestos sean erróneos por lo que será necesario revisar las conclusiones obtenidas. Este tipo de razonamiento que detalla que al introducir nuevos axiomas se puedan invalidar deducciones ya realizadas se le conoce como lógica *no-monótona* (del inglés Non-Monotonic Logic [172]).

2.8.3. Universo Cerrado versus Universo Abierto

Relacionado a los conceptos anteriores, se debe de tener en cuenta la completitud de la información que se dispone cuando se utiliza un mecanismo de inferencia.

Cuando de antemano se conoce que la información disponible es completa, se trabaja en un *Universo Cerrado* (CWA - Closed World Assumption) [155]; la información necesaria está contenida en el sistema (se asume que se conoce todo a cerca del mundo - *the world is closed*), la ausencia de un hecho es equivalente a que el mismo es falso, o dicho de otra manera, todo lo que no se conoce (no se puede demostrar) es falso.

Aunque este mecanismo de razonamiento monótono es consistente y confiable, en muchos casos no tiene aplicación en el mundo real, ya que en muchos casos se desea tener capacidades de inferir conocimiento en un mundo en donde la información

disponible es incompleta; este escenario se conoce como *Universo Abierto* (OWA - Open World Assumption) [155].

En un *Universo Abierto* se establece que todo lo que no se conoce es indefinido, admitiendo que el conocimiento del mundo es incompleto. Al agregar nuevo conocimiento, no se invalidan las deducciones que ya existen; es el enfoque opuesto al *Universo Cerrado*.

Podemos resumir los conceptos anteriores de la siguiente forma:

```
Open World Assumption  → No Completo → Monótono
Close World Assumption → Completo    → No-Monótono
```

Los motores de evaluación de reglas asumen la existencia de un *Universo Cerrado* (CWA) sobre el cual se trabaja.

Hasta aquí se resumen diferentes aspectos relacionados con el estado del arte en lo que tiene que ver con la base teórica y fundamentos principales que sustentan el enfoque de administración de reglas de negocio [204]. A partir de la sección 2.9 se presenta una revisión del mercado de reglas de negocio existente.

2.9. Mercado

El mercado de reglas de negocio es un sector volátil y prueba de ellos son las adquisiciones que se han realizado a partir del año 2007 y durante el 2008 [128].

Tres tendencias fundamentales apuntan al uso masivo de estas tecnologías [210].

- Creación de herramientas y procesos que permitan a los analistas de negocio crear y mantener sus reglas.
- Vendedores independientes están agregando aplicaciones que integran el uso de motores de evaluación de reglas como forma de distinguirse de empresas como IBM[8], Microsoft[9], Oracle[10], SAP[11] y otros vendedores de plataformas que incluyen opciones de reglas de negocio en su stack de tecnología.
- Una nueva ola de consolidación de mercado apostando a este tipo de tecnología[12].

Se detalla un resumen de las previsiones que realizan las consultoras IDC [31], Gartner [24] y Forrester [22] sobre el área de estudio.

[8] http://www.ibm.com
[9] http://www.microsoft.com
[10] http://www.oracle.com
[11] http://www.sap.com
[12] http://www.idc.com/getdoc.jsp?containerId=214643

2.9.1. IDC

La consultora IDC (International Data Corporation) publica periódicamente estudios relacionados al área de administración de reglas de negocio. En *Worldwide Business Rules Management Systems - 2009–2013 Forecast Update and 2008 Vendor Shares*[13] se analiza el mercado de reglas de negocio en el período 2006-2008 y realiza una previsión para el período 2009-2013.

En el período 2006-2008 las herramientas que ofrecen una solución de reglas de negocio líderes del mercado son:

- IBM (ILOG) [30].
- FICO (Fair Isaac) [21].
- CA (Aion) [10].
- Pegasystems (Pegarules) [87].
- Oracle (Haley) [66].

El mercado de reglas de negocio ha madurado y se establece como una herramienta clave en la construcción de aplicaciones aptas al cambio. Existen una serie de razones incluyendo:

- La alineación del negocio y las Tecnologías de la Información para aumentar la agilidad.
- La automatización y mejora en la administración de procesos de negocio.

A pesar de la recesión existente, el segmento de mercado relacionado con las reglas de negocio registra un crecimiento sostenido. Lo confirman recientes adquisiciones de la tecnología realizada por empresas como IBM (adquiriendo ILOG) y Oracle (adquiriendo Haley). Debido a esto, se estima que las perspectivas a largo plazo del mercado de reglas de negocio siga siendo positiva y que las últimas adquisiciones realizadas contribuyan a aumentar la visibilidad de éste mercado.

2.9.2. Forrester Research

La consultora Forrester en relación al área de administración de reglas de negocio publica regularmente lo que le denomina *The Forrester Wave - Business Rules Platforms* donde analiza la oferta y los líderes en este nicho de mercado.

El 8 de Abril de 2008 se publicó el análisis actualizado al segundo cuarto del 2008 en donde se evaluaron 13 plataformas de reglas de negocio de 11 proveedores utilizando 175 diferentes criterios y encontró que *IBM ILOG* [30], FICO Blaze Advisor [21] y

[13] `http://www.idc.com/research/viewtoc.jsp?containerId=220589`

Pegasystems PegaRULES [87] son los líderes dentro de las plataformas de reglas de negocio[14].

Para poder entender mejor las plataformas de reglas de negocio, Forrester creó 5 vistas del mercado:

- *Plataformas de propósito general*: herramientas que ofrecen una solución de reglas de negocio para plataformas heterogéneas. Tal como detalla la Figura 2.7, las herramientas líderes del mercado son WebSphere ILOG JRules, FICO Blaze Advisor y Pegasystems.

Figura 2.7.: Líderes en el mercado de reglas de negocio

- *Plataformas especializadas*: herramientas específicas al ámbito financiero, como por ejemplo *Predigy* [69] y *Experian* [17], no tomadas en cuenta para este resumen.

- *Plataformas para Java*: herramientas que ofrecen una solución de reglas de negocio para la plataforma Java. Tal como detalla la Figura 2.8 los principales exponentes son *WebSphere ILOG JRules* y *Corticon Technologies*.

[14] `http://www.forrester.com/go?docid=39088`

Figura 2.8.: Líderes en el mercado de reglas de negocio para Java

- *Plataformas para .Net*: herramientas que ofrecen una solución de reglas de negocio para la plataforma .Net. Para el caso .Net se verifica que *IBM ILOG* con su producto *Rules for .Net* tiene ventaja competitiva con respecto a sus competidores, tal como se detalla en la Figura 2.9.

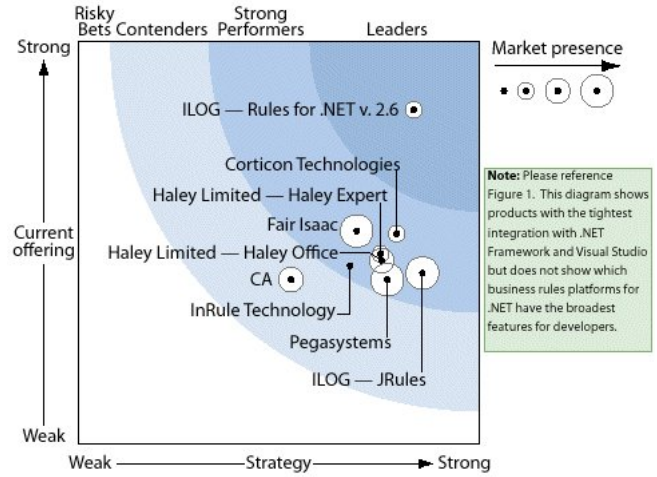


Figura 2.9.: Líderes en el mercado de reglas de negocio para .Net

- *Plataformas para analistas de negocio*: herramientas con capacidades de administración de reglas de negocio. Se verifica que hay una paridad entre varios de los proveedores de soluciones de reglas de negocio en lo que refiere al desarrollo de tecnologías que apoyen en todo el ciclo de vida de las reglas de negocio. Los principales exponentes son *Corticon*, *Pegasystems*, *IBM ILOG* y

Haley, tal como detalla la Figura 2.10. Es importante resaltar que la empresa *Haley Limited* fue adquirida por *Oracle* en Octubre de 2008.

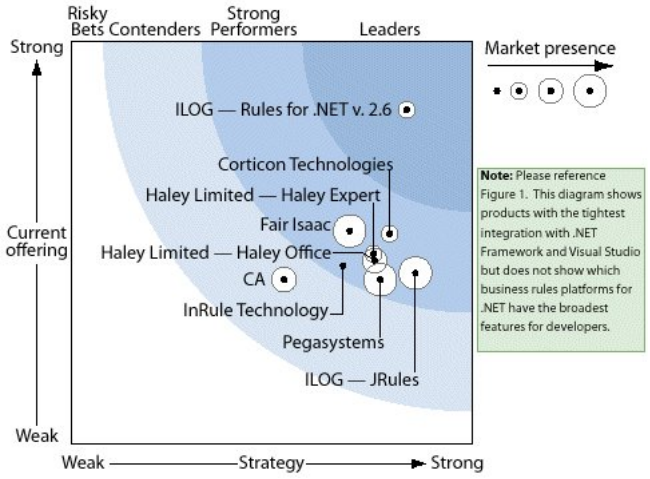


Figura 2.10.: Líderes en el mercado de Administración de reglas de negocio

Forrester ha identificado a las plataformas de reglas de negocio como estratégicas para permitir aplicaciones de negocio dinámicas porque habilitan la construcción de aplicaciones flexibles aptas al cambio.

2.9.3. Gartner

La consultora Gartner analiza a las herramientas con capacidades de administración de reglas de negocio en el contexto de las herramientas BPM (Gestión de Procesos de Negocio) [139]. En *Hype Cycle for Business Process Management*[15] destaca que cada vez más las herramientas BPM tienen capacidades de administrar todos los aspectos de un proceso – personas, máquinas, información, reglas de negocio y políticas. Es en este contexto es donde se analiza al enfoque de administración de reglas de negocio.

La tecnología que sustenta a los motores de evaluación de reglas (BRE - Business Rule Engine) [135] no es nueva y los componentes fundamentales no han cambiado desde hace algún tiempo. Ocasionalmente, aparecen mejoras o nuevos algoritmos relacionados con el procesamiento de reglas, pero en general el núcleo de ésta tecnología es estable.

Sin embargo, se está innovando en la forma de manipular las reglas de negocio a través de los gerenciadores de reglas (BRMS - Business Rule Management System) [221]. Dichos componentes administran en forma completa el ciclo de vida de una regla; desde su concepción en lenguaje natural, modelado y almacenamiento en un

[15] `http://www.gartner.com/DisplayDocument?doc_cd=149186`

repositorio de reglas, validación, verificación y posterior ejecución en un ambiente de producción.

La tecnología BRE se ha utilizado en forma extensiva en varias industrias como por ejemplo en seguros y servicios financieros, sin embargo no logró capitalizarse su uso en forma más general por la industria de Software.

Debido al posicionamiento de la tecnología BPM y la aparición del estilo arquitectónico de orientación a servicios (SOA - Software Oriented Architecture) [144], se estima que más énfasis se dará en explicitar reglas de negocio para lograr componentes reutilizables.

Gartner declara que la administración efectiva de políticas y reglas empresariales en el negocio puede llegar a tener un alto impacto, como por ejemplo:

- Aumentar el conocimiento explícito y rastrear puntos de decisión subyacentes en procesos de negocio crítico a partir de reglas y políticas de negocio.
- Reducir el tiempo de reacción ante cambios en puntos de decisión.
- Aumentar la agilidad, consistencia y rigor de puntos de decisión empresariales, devolviendo el control primario de las decisiones automatizadas al experto del negocio.

2.9.4. Criterios de búsqueda en Internet

Con el objetivo de constatar el interés sobre la tecnología de reglas de negocio que indican las consultoras (analizado en 2.9.1, 2.9.2 y 2.9.3), se realiza una investigación informal en Internet utilizando dos servicios existentes: *Indeed* y *Google Trends* para obtener algún tipo de indicador que permita constatar que la predicción que realizan las consultoras es tangible.

- *Google Trends*[16] es un servicio provisto por *Google* que compara la tendencia de búsqueda de diferentes tópicos, permite conocer las palabras clave más utilizadas en *Google* y realizar comparaciones. *Google Search*[17] mantiene el liderazgo en lo que refiere a los motores de búsqueda mas utilizados[18], por lo que utilizando *Google Trends* se obtiene información representativa sobre tendencias de búsqueda.
- *Indeed*[19] es uno de los 10 motores de búsqueda específico para trabajos (comúnmente denominado *job search engine* [129]) más utilizados. Permite encontrar oportunidades\ofertas de trabajo realizadas por miles de empresas

[16] `http://www.google.com/trends`
[17] `http://www.google.com/`
[18] `http://www.statowl.com/search_engine_market_share.php`
[19] `http://www.indeed.com/jobtrends`

en todo el mundo. Con esta herramienta interesa conocer si ya existen ofertas en el mercado específicas para trabajar con motores de reglas y particularmente cómo evoluciona en los últimos tiempos.

Se comienza definiendo un conjunto de palabras clave que identifican al área de estudio. Las mismas se utilizarán para realizar búsquedas en los sitios detallados anteriormente para conocer el interés por las tecnologías relacionadas.

- *Business Rules Engine*: para tener un estimativo del grado de interés en motores de evaluación de reglas.

- *BRMS*: se busca por el componente de gerenciamiento de reglas de negocio que ofrece capacidades de administración de reglas y complementa a los motores de evaluación.

Búsqueda "Business Rules Engine"

En el sitio de *Google Trends* se realiza una búsqueda por el texto "*business rules engine*" obteniéndose el gráfico de la Figura 2.11; que representa con cuánta frecuencia se realiza una búsqueda por las palabras clave *business rules engine*.

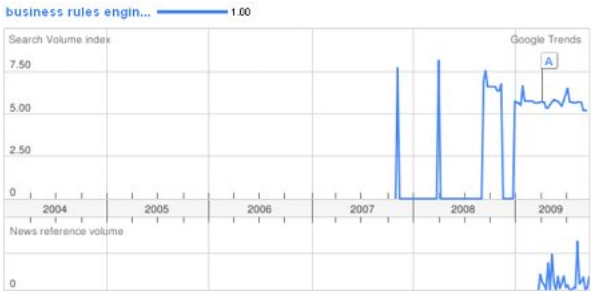


Figura 2.11.: Búsqueda por "business rules engine" en Google Trends

El eje horizontal de la gráfica representa el tiempo, y el eje vertical representa la frecuencia con la que se ha buscado el término globalmente.

Por su parte, se accede al sitio de *Indeed* y realiza una búsqueda por las mismas palabras clave, obteniendo el resultado de la Figura 2.12. El resultado representa la frecuencia con la cual aparecen ofertas de trabajo que incluyan las palabras clave *business rules engine*.

Figura 2.12.: Búsqueda por "business rules engine" en Indeed

Búsqueda "BRMS"

Nuevamente, en el sitio de *Google Trends* se realiza una búsqueda por las palabras clave "*business rules management system*" detallándose el siguiente mensaje:

> *business rules management system - do not have enough search volume to show graphs.*

por lo que se modifican las palabras clave utilizando el acrónimo relacionado "*BRMS*" y se realiza una nueva búsqueda. El resultado se detalla en la Figura 2.13.

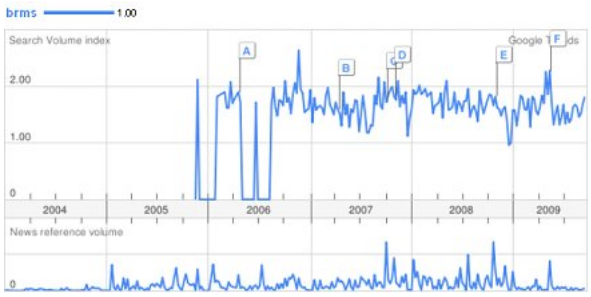


Figura 2.13.: Búsqueda por "BRMS" en Google Trends

La misma búsqueda en el sitio *Indeed* para las dos frases se detallan a continuación. El caso *BRMS* se detalla en la Figura 2.14.

Figura 2.14.: Búsqueda por "BRMS" en Indeed

El caso "*Business Rules Management System*" se detalla en la Figura 2.15.

Figura 2.15.: Búsqueda por "Business Rules Management System" en Indeed

Informalmente se concluye que las predicciones realizadas por las consultoras son correctas; existe un interés y adopción creciente en lo que respecta a esta tecnología según búsquedas realizadas en *Google* (en el período 2006-2009), y a su vez se verifica un incremento tangible en las ofertas laborales según el sitio *Indeed*.

2.9.5. Movimiento de Productos

El interés sobre la tecnología de administración de reglas de negocio (detallado en las secciones 2.9.1 a 2.9.4) también se refleja en adquisiciones que se han realizado desde 2007, en donde empresas como *Oracle*, *SAP* e *IBM* acceden al mercado de reglas de negocio, ya sea adaptando su tecnología o adquiriendo productos ya existentes.

A continuación se detalla un resumen del movimiento de productos del mercado de motores de evaluación de reglas de negocio.

- Drools (motor de reglas de negocio para la plataforma Java) pasa a formar parte de JBoss [39] (*Octubre 2005*).
- FICO (principal proovedor de herramientas relacionadas con la administración de reglas de negocio) adquiere RulesPower [19] (*Setiembre 2005*).
- Trilogy (empresa consultora) adquiere Versata [102] (*Diciembre 2005*).

- Red Hat (compañía responsable de la creación y mantenimiento de una distribución del sistema operativo GNU/Linux) adquiere JBoss [76] (*Abril 2006*).
- SAP (uno de los principales proveedores de software empresarial) adquiere YASU Technologies [91] (*Octubre 2007*).
- Haley Systems es adquirido por RuleBurst Limited [20] (*Noviembre 2007*).
- IBM (una de las empresas de servicios basados en tecnología de información más grande del mundo) adquiere ILOG [30] (*Julio 2008*), uno de los principales proovedores de herramientas de administración de reglas de negocio.
- El grupo Bosch (compañía alemana) adquiere Innovations Software Technology [3] (*Setiembre 2008*).
- Oracle (una de las mayores compañías de software del mundo) adquiere Haley [65] (*Octubre 2008*).
- Savvion (una empresa con desarrollos relacionados con la administración de procesos) disponibiliza tecnología de gestión de reglas de negocio [92] (*Noviembre 2008*).

2.10. Estándares

Si bien existen varios estándares en relación a la especificación de reglas de negocio, actualmente (Marzo/2010) se verifica que no han sido adoptados por la industria, en el sentido de que cada herramienta existente en el mercado desarrolla su propia especificación e implementación de reglas de acuerdo a las diferentes funcionalidades que ofrece.

Existen esfuerzos importantes para estandarizar un lenguaje de reglas de negocio llevados adelante principalmente por la W3C [110] con estándares como OWL [68] y RIF [82], la OMG [57] con estándares como SBVR [93] y PRR [71] y consorcios independientes como es el caso de RuleML [86].

En octubre de 2008, en la conferencia *Business Rules Forum* [8], hubo un panel de vendedores en donde, entre otras cosas, se discutió acerca de los emprendimientos para estandarizar lenguajes de reglas y en lo que respecta al corto plazo, se estima que no existan avances representativos en el tema [220].

La consultora Gartner [24] analiza este problema [212] detallando que no existe una forma de administrar reglas de negocio con diferentes motores de evaluación de reglas y tecnologías porque las representaciones de reglas son totalmente diferentes, no permitiendo capturar el contexto de una regla. En términos generales apuesta

al estándar SBVR [93] debido a que está enfocado a vocabularios y significado, que combinado con el concepto de ontología [157], lograría un compromiso ideal para mejorar la administración de reglas.

Con respecto a la industria, la mayoría de las herramientas del mercado utilizan variaciones de lo que son las *reglas de producción* representadas por el estándar PRR [71].

Aquí se realiza una breve reseña de los estándares más importantes y lenguajes de definición de reglas de herramientas comerciales. Para profundizar consulte el Anexo C.

2.10.1. RuleML

Rule Markup Language [86]; es un lenguaje basado en XML [18] para especificar reglas.

Su objetivo es crear un conjunto de lenguajes de marcado neutral para lograr interoperabilidad semántica entre diferentes motores de reglas de negocio. Para ello, sus fundadores han elegido enfocarse en intercambios entre estándares de la industria en vez de prototipos académicos.

Es un estándar que consta de una gran gama de sublenguajes que están evolucionando permanentemente. Su extensibilidad permite que nuevos sublenguajes se puedan agregar con facilidad.

Dada la importancia que se le da a la interacción entre sistemas, se están desarrollando traductores entre diferentes motores de reglas de negocio y RuleML.

2.10.2. RIF

Rule Interchange Format [82]; es un estándar de la W3C para proveer un lenguaje basado en XML para el intercambio de reglas entre ambientes de ejecución, por un lado entre reglas lógicas y de producción y también para interoperar con tecnologías relacionadas con Web semántica utilizando RDF, RDF Schema y OWL. La especificación alcanzó el status de candidato a recomendación en octubre de 2009 [80].

Tiene como objetivo lograr establecer un lenguaje común para poder mapear diferentes lenguajes de reglas, permitiendo que las reglas específicas para un producto y plataforma se puedan compartir y reutilizar por otras aplicaciones que usen otros motores de evaluación de reglas.

RIF está diseñado como una familia de dialectos en donde cada uno es una colección de componentes que trabajan como un todo, formando una interlingua. Los dialectos son Basic Logic Dialect [78] un formato de reglas lógicas, Production Rules Dialect [81] para representar reglas de producción y Core [79] detallando el conjunto común de los dos dialectos anteriores.

2.10.3. SBVR

Semantics of Business Vocabulary and Rules [93] está formalmente definido como una taxonomía que describe operaciones y reglas de negocio elementales. Es una especificación propuesta inicialmente por la Business Rule Group [100] y tomada por la OMG [57] en su meta modelo expresado en UML [103], integrado dentro de lo que se denomina Model-Driven Architecture (MDA) [191].

Detalla cómo las reglas y vocabulario de negocio se pueden capturar y especificar en términos no ambiguos en un lenguaje lógico formal con expresiones formuladas en un lenguaje natural restringido.

Por vocabulario nos referimos a todas aquellas palabras y términos que representan conceptos y operaciones en una organización.

Se enfoca principalmente a los expertos de negocios, no al procesamiento automático de las reglas; es decir, es para que los analistas de negocio puedan modelar el propio negocio, usando sus propios términos, independiente del manejo implícito o explícito que se haga desde los *sistemas de información*.

2.10.4. PRR

Production Rule Representation [71] es un estándar de la OMG para proveer un meta modelo independiente de la plataforma (PIM – Platform Independent Model). Define un modelo formal para las reglas de producción.

Se compone de:

- Una estructura núcleo referida como *PRR Core* en donde se definen las reglas de producción para utilizarse en inferencia hacia adelante o procesamiento secuencial.
- Una extensión opcional que permite utilizar OCL [56] para definir el contenido semántico de las reglas.

Se prevee la existencia de un mecanismo de correlación con el estándar RIF como mecanismo que permita intercambiar definiciones de reglas ejecutables entre modelos.

2.10.5. JSR-94

La especificación Java™ Rule Engine API (JSR 94) [36] desarrollado por la Java Community Process (JPC)[20], define una API Java para interactuar con motores de reglas desde la plataforma Java. La iniciativa JSR 94 es un intento de unificar en una única API[21] las diferentes opciones existentes en el mercado; por lo tanto, la

[20] `http://jcp.org`
[21] Application Program Interface

misma provee un *mínimo denominador común* de las funcionalidades de los diferentes motores de reglas.

El emprendimiento no funcionó debido a que no se ofrece suficiente funcionalidad desde el punto de vista de la API en sí y en general los proveedores hicieron sólo modificaciones parciales a sus productos para soportar la nueva interfaz definida.

2.10.6. OWL y OWL 2

Ontology Web Language [68] es un lenguaje de marcado para publicar y compartir datos usando ontologías [157] en la web. OWL tiene como objetivo facilitar un modelo de marcado construido sobre RDF y codificado en XML.

OWL tiene tres variantes:

- *OWL Lite*: soporta la clasificación de jerarquías y definición de restricciones simples.
- *OWL DL*: permite una mayor expresividad reteniendo completitud computacional en donde se asegura que todas las conclusiones son computables y decidibles en tiempo finito.
- *OWL Full*: Máximo nivel de expresividad sin garantía de computabilidad.

Estas variantes incorporan diferentes funcionalidades, y en general, *OWL Lite* es más sencillo que *OWL DL* y *OWL DL* es más sencillo que *OWL Full*.

OWL Lite está construido de tal forma que toda sentencia pueda ser resuelta en tiempo finito. La versión más completa de *OWL DL* puede contener 'bucles' infinitos.

OWL 2 [68] refina y extiende OWL; la especificación alcanzó el status de candidato a recomendación en junio de 2009 [67]. OWL 2 corrige varios problemas de la especificación inicial y agrega expresividad al lenguaje acorde a las últimas investigaciones relacionadas con DL (Description Logics) [117].

Además, agrega la posibilidad de utilizar XML y otros formatos compactos de representación para el intercambio de ontologías [157]. Finalmente, OWL 2 ofrece 3 subconjuntos del lenguaje denominados perfiles:

- *OWL 2 EL* es particularmente útil para ontologías con gran cantidad de ítems y propiedades; el razonamiento es computable en tiempo polinómico.
- *OWL 2 QL* es de interés para ontologías con una importante cantidad de datos en donde el mecanismo de consulta puede referenciar bases de datos relacionales. Se basa en *DL-Lite*.
- *OWL 2 RL* es un fragmento de OWL 2 que permite el uso de tecnologías basadas en reglas.

2.10.7. OCL

Object Constraint Language [56] es un lenguaje para la descripción formal de expresiones en los modelos UML. Sus expresiones pueden representar invariantes, pre-condiciones, post-condiciones, inicializaciones, guardas, reglas de derivación, así como consultas a objetos para determinar su estado.

Su papel principal es el de completar los diferentes artefactos de la notación UML con requerimientos formalmente expresados; la versión OCL2.0 (Object Constraint Language 2.0) fue adoptado en octubre de 2003 por el grupo OMG como parte de UML 2.0.

2.11. Lenguajes Comerciales

En el contexto comercial, cada motor de evaluación de reglas implementa sus reglas de negocio de la manera que más le convenga para soportar sus funcionalidades. Existen diferentes formas de especificar las reglas de negocio con sus diferentes lenguajes, como por ejemplo:

- Lenguajes orientados a objetos como son los utilizados por las herramientas Drools [13], JRules [107], Blaze Advisor [21] y Oracle Business Rules [66].
- Web semántica como es el caso JenaRules [41].
- Inteligencia artificial utilizando frames[22] como es el caso de Jess [44] y Clips [11].
- Internal DSL[23] como es el caso de RuleBy [85].

A partir del análisis de mercado realizado y teniendo en cuenta la evaluación de diferentes motores se realiza un resumen de los principales lenguajes de definición de reglas propietarios.

2.11.1. RL

Rule Language[24], lenguaje propietario de Oracle para definir reglas de negocio específico para trabajar desde el ambiente Java.

Las reglas siguen una estructura *If-Then* en donde se definen las condiciones y acciones respectivamente.

Ejemplo:

[22] `http://www.cse.unsw.edu.au/~billw/aidict.html#frame`
[23] `http://martinfowler.com/dslwip/InternalOverview.html`
[24] `http://download.oracle.com/docs/cd/B25221_04/web.1013/b15985.pdf`

```
rule driverAge
{
  if (fact Driver d1 && d1.age < 16)
  {
  println("Invalid Driver");
  }
}
```

A partir de la versión 11g del producto Middleware Oracle Fusion[25], se agregan interfaces de alto nivel para poder definir las reglas.

2.11.2. SRL

Structured Rules Language, es un lenguaje propietario basado en XML desarrollado por la empresa FICO utilizado en su producto Blaze Advisor [21].

Es un lenguaje orientado a objetos diseñado con el objetivo de lograr escribir y leer reglas de negocio de forma "similar" al inglés; por ejemplo:

```
If
  the applicant for the loan is a current bank customer
  and has a checking account with direct paycheck deposit
  and has a balance greater than $10,000
then
  lower the interest rate by 1%.
```

2.11.3. DRL

Drools Rule Language [13], lenguaje de definición de reglas utilizado por la herramienta Drools. Las reglas siguen una estructura *When-Then* en donde se definen las condiciones y acciones respectivamente.

Ejemplo:

```
rule "Check Patient for Heart Attack"
when
  $patient : Patient(
    breathing == false ,
    pulse == false)
    (face == blue or face == white)
  )
then
    $hospital.callForAssistance();
    $patient.startCPR();
end
```

[25] http://www.oracle.com/lang/es/products/middleware/index.html

2.11.4. JessML

Es el lenguaje propietario de definición de reglas utilizado por Jess [44]. Primero se definen los objetos y propiedades sobre los cuales se desea trabajar y posteriormente se definen las reglas en base a dichos objetos.

Ejemplo de una regla:

```
( defrule cambiar-bebe-si-esta-mojado
  ?mojado <- (bebe-mojado)
  =>
  (cambiar-bebe)
  (retract ?mojado)
)
```

2.11.5. BAL

Business Action Language, lenguaje de reglas de negocio de alto nivel de abstracción utilizado por la herramienta JRules [107]. Mapea un conjunto de verbos y sustantivos usado por los analistas de negocio en reglas que utilizan las clases, métodos y campos del modelo de objetos del negocio.

Ejemplo:

```
If
    the shopping cart value is greater than $ 100
    and the customer category is gold
then
    Apply a 15% discount
```

2.11.6. IRL

Ilog Rule Language representa el formato ejecutable del lenguaje de reglas utilizado por IBM ILOG.

Ejemplo:

```
rule latestBankruptcy1 {
  property task = "computation";
  when {
    borrower: Borrower(hasLatestBankrupcy()) from borrower;
    loan: Loan() from loan;
    evaluate (age: DateUtil.getAge(borrower.getLatestBankruptcyDate(),
                                   loan.getStartDate()));
    evaluate (age <=1);
  }
  then {
    report.addCorporateScore(-200);
  }
}
```

2.11.7. JenaRules

JenaRules [41] se basa en RDF(S) y utiliza la representación en tripletas definido por RDF (también permite especificar las reglas en notación N3[26] y Turtle[27]), forma parte de la herramienta Jena utilizada para la Web Semántica en la plataforma Java. Para poder trabajar con JenaRules es necesario transformar los objetos del negocio a representación RDF para poder utilizar los motores de inferencia provistos.

Una clase Java mapea a la definición *rdfs:Class*; mientras que las propiedades de las clases mapean a la propiedad *rdf:Property*. Las propiedades RDFS *rdfs:domain* y *rdfs:range* especifican la *clase* y *tipo* de propiedad.

Se tiene la clase *Person* y *Driver* con la siguiente especificación:

```
public class Person
  private String name;
  private int age;

public class Driver extends Person
  private int accidentsNumber;
  //other properties and/or operations
```

La representación RDF es la siguiente:

```
<rdfs:Class rdf:about="Person" />
<rdfs:Class rdf:about="Driver">
  <rdfs:subClassOf rdf:resource="#Person"/>
</rdfs:Class>
<rdf:Property rdf:about="Person.name">
  <rdfs:range rdf:resource="xs:string"/>
  <rdfs:domain rdf:resource="#Person"/>
</rdf:Property>
<rdf:Property rdf:about="Person.age">
  <rdfs:range rdf:resource="xs:int"/>
  <rdfs:domain rdf:resource="#Person"/>
</rdf:Property>
<rdf:Property rdf:about="Driver.accidentsNumber">
  <rdfs:range rdf:resource="xs:int"/>
  <rdfs:domain rdf:resource="#Driver"/>
</rdf:Property>
```

Un ejemplo de una regla que utiliza las clases detalladas anteriormente es la siguiente:

```
@prefix eg: http://com.sample/
# Hello World
[HelloWorld:(?m rdf:type eg:Person)
```

[26] http://www.w3.org/DesignIssues/Notation3
[27] http://www.w3.org/TeamSubmission/turtle/

```
(?m eg:name 'Miguel'^^xsd:string)
(?m eg:age ?s)
→
(?m eg:name 'Michael^^xsd:string)
log(?m, 'HelloWold rule has matched', 'Changing Person Name')
```

que define que cuando exista una persona de nombre “Miguel”, se registra el caso a través de un mensaje de log.

2.12. Revisión de Herramientas

Existe una cantidad importante de herramientas en el mercado tanto de código abierto como comerciales. Se detalla un breve resumen de las herramientas de mayor relevancia de ambos tipos (código abierto y comercial).

Se toma como referencia el análisis de mercado realizado en el Anexo B.

2.12.1. Herramientas Open Source

Las principales herramientas de código abierto son: *Drools* [13], *Clips* [11], *NxBRE* [55], *Ruleby* [85] y *Jena* [40].

Drools

Drools [13] es un motor de inferencia basado en reglas de tipo inferencia hacia adelante para Java.

Fue adquirido por *JBoss* y pasó a llamarse *JBoss Rules*[28] incluido dentro de lo que se denomina JEMS (JBoss Enterprise Middleware Systems[29]); se clasifica dentro de los motores de ejecución de reglas como Production Rule System.

Provee un lenguaje declarativo lo suficientemente flexible para especificar la semántica de cualquier dominio con varios lenguajes específicos de dominio.

Existe una implementación para la plataforma .Net denominado *Drools.NET* [14] soportando la mayoría de las funcionalidades de *Drools 3.0*.

Características:

- Utiliza lenguajes específicos de dominio para que tanto los analistas de negocio como desarrolladores puedan detallar las reglas en forma natural y provee deferentes interfaces para su definición.

[28] `http://www.jboss.com/products/platforms/brms/`
[29] `http://www.jboss.com/products/index`

- Administra las reglas de negocio en forma independiente del código de la aplicación.
- Incluye un complemento para usarse de forma integrada a Eclipse.

Clips

Clips (C Language Integrated Production System) [11], es una herramienta de dominio público utilizado para construir sistemas expertos.

La sintaxis y nombre fue inspirado por Charles Forgy. La primera versión fue desarrollada en 1985 en la NASA como una alternativa a un sistema experto ya existente denominado ART*Inference. Incorpora un lenguaje orientado a objeto denominado COOL.

Existen varios descendientes del lenguaje Clips como por ejemplo Jess [44], FuzzyClips [23], EHSIS[30].

NxBRE

NxBRE [55] es un motor de evaluación de reglas de negocio open-source implementado en .Net basado en JxBRE [48].

Ofrece dos aproximaciones diferentes para utilizarlo: como motor de inferencias o como motor de flujo.

La definición de reglas se puede utilizar con la notación RuleML [86] o en un formato propietario.

RuleBy

Ruleby [85] es un motor de reglas desarrollado en Ruby [83], es probablemente uno de los más populares y maduros en esta plataforma.

Provee un DSL [169] para definir e insertar las reglas (de tipo regla de producción) en la memoria de trabajo del motor de evaluación; los hechos son los propios objetos Ruby. El motor de inferencia utiliza el algoritmo *Rete* totalmente implementado en Ruby.

Jena

Jena [40] es un framework open source (licencia BSD[31]) para la Web Semántica, originario de HP Labs[32]. Si bien incluye un motor de reglas genérico, está enfocado fuertemente dentro del marco de la Web Semántica, proveyendo un ambiente programático para interactuar con RDF, RDFS, OWL y SPARQL [96].

[30] `http://erabaki.ehu.es/ehsis/`
[31] `http://www.opensource.org/licenses/bsd-license.php`
[32] `http://www.hpl.hp.com/semweb/`

Incluye diferentes razonadores de los cuales interesa el de propósito general ya que es un razonador basado en reglas. Soporta razonamiento hacia adelante, atrás y una estrategia híbrida de inferencia.

2.12.2. Herramientas Comerciales

Las principales herramientas comerciales son: *Oracle Rules Engine* [66], *Jess* [44], *InRule* [32], *MsBRE* [101], *Blaze Advisor* [21], *WebSphere ILOG JRules* [107], *Rules for .Net* [108], *Visual Rules* [105] y *SAP Netweaver BRM* [90].

Oracle Rules Engine

Oracle Rules Engine [66] está conformado por un motor de inferencia, un lenguaje de reglas propietario denominado *Rule Language* y una herramienta para la edición de reglas.

Componentes principales:

- *Rule Author*: se utiliza mediante un browser para crear y editar reglas de negocio.
- *Rule language* (RL): es el lenguaje de programación de reglas integrado a Java.
- *Motor de ejecución*: es un derivado de *Jess*, utiliza una implementación del algoritmo *Rete*.
- *Rule SDK*: consta de una API para que el desarrollador pueda definir objetos del negocio, ejecutar y monitorear las reglas.
- Compatible con el estándar *JSR-94*.

Jess

Jess [44] es un motor de reglas desarrollado en los laboratorios Sandia National Laboratories[33]. Características principales:

- Las reglas se escriben utilizando notación del tipo Lisp [113] o Prolog [216].
- Inicialmente derivó de Clips y se reescribe en Java.
- Utiliza un lenguaje propio de definición de reglas denominado *jessML*.
- Utiliza una versión mejorada del algoritmo Rete [150].

[33] `http://www.sandia.gov/`

InRule

InRule [32] es un motor de reglas de negocio que extiende Windows Workflow Foundation [109] desarrollado por InRule Technology, específico para la plataforma .Net. Provee tecnología de reglas de negocio para la autoría, verificación y administración centralizada de reglas.

Principales características:

- Facilidad para la creación de reglas, mediante herramientas que permiten a los desarrolladores y analistas de negocio crear, testear y mantener reglas.
- Inspección en tiempo real del resultado de la ejecución de reglas.
- Creación de la interfaz de usuario en forma dinámica basado en las reglas.

Microsoft Business Rules Engine

MsBRE [101] es un motor de ejecución de reglas de tipo razonamiento hacia adelante incluido en la distribución de Biztalk [101]. Consta de los siguientes componentes:

- *Business Rules Composer*: para construir las políticas de negocio; provee interfaces para que los desarrolladores, administradores y analistas de negocio puedan construir las restricciones de negocio necesarias.
- *Rules Engine Deployment Wizard*: para poner en producción las políticas creadas.

El repositorio de reglas centraliza las políticas y vocabularios, se almacena en una base de datos SQL Server[34].

Blaze Advisor

Blaze Advisor [21] es una herramienta desarrollada por la empresa *FICO*. Ofrece un entorno para crear, probar y desarrollar reglas de negocio.

Características:

- Administración del repositorio de reglas centralizado pudiéndose almacenar en archivos XML, bases de datos o directorios LDAP[35].
- Soporte de tablas y árboles de decisión para la representación de reglas de negocio en forma compacta.

[34] `http://msdn.microsoft.com/en-us/sqlserver/default.aspx`
[35] `http://www.faqs.org/rfcs/rfc2251.html`

- Seguimiento y versionado de reglas.
- Utiliza un lenguaje propio de definición de reglas denominado Structured Rules Language y un vocabulario independiente de la plataforma denominado Business Object Model Adapter.
- Repositorio único para múltiples plataformas de ejecución nativas (Java, .NET, COBOL), pudiendo utilizarse también desde otros ambientes utilizando Web Services.
- Soporta múltiples modos de ejecución: inferencia (hacia adelante, hacia atrás), secuencial y secuencial compilado.
- Brinda un framework para test unitario de reglas de negocio denominado *brUnit*.

WebSphere ILOG JRules

WebSphere ILOG JRules [107] es un motor de evaluación de reglas de negocio para Java desarrollado por la empresa *IBM ILOG*. Consta de los siguientes componentes:

- *Rule Builder*: Es un entorno gráfico para el desarrollo, test y gestión de las reglas de negocio. Permite definir reglas utilizando varias opciones, una sintaxis muy similar a Java y también un lenguaje propietario de definición de reglas.
- *JRules Repository*: Es el repositorio central de las reglas de negocio.
- *Business Rule Editor*: Editor gráfico para escribir reglas de manera sencilla e intuitiva.
- *JRules Rule Server*: motor de inferencia para ser utilizando en un entorno J2EE\J2SE [35, 37]. El motor de JRules consta de un conjunto de clases java con la lógica necesaria para ejecutar reglas. Representa a los objetos de la aplicación directamente con código Java sin la necesidad de interpretar código en tiempo de ejecución.

Rules for .Net

Rules for .Net [108] es un motor de evaluación de reglas de negocio para la plataforma .Net 3.0 integrado con Microsoft Windows Workflow Foundation [109] desarrollado por la empresa IBM ILOG.

Se dispone de un repositorio accesible vía Sharepoint Services [53], un conjunto de complementos para Visual Studio[36] y Microsoft Word.

[36] `http://msdn.microsoft.com/en-us/vstudio/default.aspx`

Los programadores se encargan de crear las reglas en Visual Studio, las exportan como documentos de tipo RuleDocs, un formato basado en XML y posteriormente los analistas de negocio las editan utilizando Word. También se dispone de complementos para Microsoft Excel para permitir crear reglas para BizTalk.

Visual Rules

Visual Rules [105] es un motor de evaluación de reglas para Java desarrollado por Innovations Software[37] con fuerte enfoque de orientación a servicios en donde las reglas se generan en código Java, pudiéndose exponer como Web Services.

QuickRules

QuickRules es un motor de evaluación de reglas de negocio para Java y .Net desarrollado por la empresa *Yasu Technologies* adquirido en el 2007 [91] por SAP.

SAP Netweaver Business Rules Management

SAP Netweaver BRM [90] es la plataforma de ejecución de reglas de negocio para su producto NetWeaver[38]; recordar que inicialmente se adquirió tecnología de *Yasu Technologies* creador de *QuickRules*.

Esta compuesto por las siguientes herramientas:

- *Compositor*: para la creación y modificación de reglas de negocio utilizando diferentes formatos como por ejemplo tablas de decisión.
- *Analizador*: permite que los expertos del negocio realicen tests, refinen, analicen y optimicen las reglas de negocio.
- *Administrador*: Interfaz web para administrar las reglas de negocio.
- *Repositorio*: Provee un ambiente para el manejo de versionado, permisos, control de acceso, alertas relacionado con las reglas de negocio.
- *Motor de ejecución*: ejecuta las reglas, integrado a la tecnología provista por SAP NetWeaver Composition Environment[39].

[37] `http://www.innovations-software.com/`
[38] `http://www.sdn.sap.com/irj/sdn/nw-70`
[39] `http://www.sdn.sap.com/irj/sdn/nw-ce`

2.13. Trabajos Académicos relacionados

En esta sección se realiza una breve reseña de algunos trabajos académicos relacionados al área de estudio. En particular, se resaltan aquellos trabajos que aportaron ideas a la propuesta que se detalla en el capítulo 3.

Existen dos proyectos de grado realizados en el INCO[40] relacionados al área de reglas de negocio que son el proyecto *Ejecutor de Reglas de Negocio* [133] y *Generador de Reglas de Negocio* [125].

El proyecto *Ejecutor de Reglas de Negocio* [133] es un proyecto de grado de la Facultad de Ingeniería en donde se analiza y experimenta el uso de reglas de negocio en la plataforma .Net utilizando el intérprete *Prolog April*[41] desarrollado en la misma institución. Para especificar las reglas de negocio se propone una extensión al estándar RuleML [86] que permite definir diferentes tipos de reglas: restricciones, activación y desactivación de reglas, especificación de cálculos. Se realiza un prototipo que provee el marco de ejecución para asegurar el cumplimiento de las restricciones que se definidan y permite modificar dichas restricciones en el ambiente de ejecución de la aplicación.

El proyecto *Generador de Reglas de Negocio* [125] es un proyecto de grado de la Facultad de Ingeniería realizado para la empresa Cabal S.A.[42]. Se desarrolla un lenguaje de reglas de negocio que soporta la definición de restricciones sobre datos, cálculos y disparo de eventos. Dichas definiciones se traducen en código Java o procedimientos almacenados que posteriormente se procesan en un evaluador de reglas.

Ambos proyectos utilizaron enfoques y tecnologías diferentes. Sin embargo, comparten un objetivo común: la especificación declarativa y separada de las aplicaciones de las restricciones clave que rigen el negocio expresadas como reglas de negocio. Los mismos se utilizaron fundamentalmente como referencia inicial para el estudio de los principales conceptos sobre los cuales se construye la teoría de reglas de negocio y como punto de partida en el análisis de herramientas.

La Universidad Nacional de Colombia[43] registra varios proyectos de investigación [115, 116] relacionados en donde utilizando mecanismos de razonamiento basados en reglas implementan agentes de software inteligentes; para ello integran en un prototipo el uso de agentes de software utilizando la plataforma JADE (Java Agent DEvelopment framework) [34] con el motor de evaluación de reglas Jess [44] para lograr agentes híbridos que exhiben comportamientos reactivos y deliberativos útiles en escenarios de simulación; en particular, analizan la negociación electrónica de contratos en el mercado de Energía Eléctrica en Colombia. En este contexto, la

[40] `http://www.fing.edu.uy/inco/`
[41] `http://www.fing.edu.uy/inco/proyectos/uyprolog/`
[42] `http://www.cabal.com.uy/`
[43] `http://www.unal.edu.co/`

variabilidad en los parámetros que rigen la negociación de los contratos se especifican como reglas de negocio que el agente de software utiliza por medio del intérprete Jess.

La Universidad Nacional Mayor de San Marcos[44] en Perú y la Universidad de Chile[45] registran trabajos relacionados al área [219, 143] detallando los puntos de contacto entre BPM (Gestión de Procesos de Negocio) y BRMS (Gestión de Reglas de Negocio). Es decir, detalla cómo las herramientas BPM cada vez más incluyen capacidades de procesamiento de reglas de negocio, como funcionalidad estándar dentro del marco de procesos de negocio. En este sentido, dichos trabajos son tomados en cuenta en el contexto del trabajo ya que es de interés poder expresar puntos de variabilidad de procesos en términos de reglas de negocio.

En varias universidades españolas, en el contexto de varios cursos relacionados con Inteligencia Artificial se evidencia al concepto de *sistemas basados en reglas* aplicado en sistemas expertos, utilizando diferentes tipos de herramientas como por ejemplo Clips [11], ART (Automated Reasoning Tool), Jess [44].

La Universidad Complutense de Madrid[46] plantea en varios cursos curriculares el uso de herramientas como Clips [11] y Jess [44] para los trabajos en laboratorio, por ejemplo en *Ingeniería de Sistemas Basados en Conocimiento*[47].

En la Universidad de Sevilla[48] utilizando ART y Clips [11] para la representación del conocimiento, en [179] se implementa un tablero semántico proposicional para comprobar si una fórmula dada es una tautología.

En la Universidad Politécnica de Madrid[49] en [147] se utiliza el marco de definición de reglas para la generación automática de programas genéticos para aplicarse a la resolución del problema de detección de lesiones de rodilla a partir de curvas isocinéticas[50]. El trabajo utilizó sistemas basados en reglas para construir un sistema capaz de detectar lesiones de rodilla analizando las curvas proporcionadas por la máquina isocinética, extrayendo conocimiento de los datos de entrenamiento.

Las Universidades de Málaga[51] y Murcia[52] registran artículos relacionados con el modelado basado en restricciones (Constraint Based Modeling) [152, 194]. En [152] se desarrolla una herramienta Web que asiste en el aprendizaje de los fundamentos de la programación orientada a objetos; se trata de un sistema inteligente

[44] `http://www.unmsm.edu.pe/`
[45] `http://www.dcc.uchile.cl/1877/channel.html`
[46] `http://www.fdi.ucm.es/`
[47] `http://www.fdi.ucm.es/profesor/belend/ISBC/isbc.html`
[48] `http://www.cica.es/`
[49] `http://www.upm.es/`
[50] una máquina isocinética registra la fuerza muscular realizada por un paciente durante la realización de un ejercicio, produciendo una serie de valores distribuidos a lo largo de la duración de dicho ejercicio.
[51] `http://www.lcc.uma.es/`
[52] `http://www.um.es/informatica/index.php`

construido siguiendo la técnica de Modelado Basado en Restricciones. El sistema elige dinámicamente el problema más adecuado al nivel de conocimiento de cada individuo. En [194] se presenta una estrategia para obtener de modo sistemático el modelo de casos de uso y conceptual a partir del modelado del negocio basado en diagramas de actividad UML. Las reglas de negocio son identificadas e incluidas en el modelo como restricciones que se tienen que satisfacer.

También, el centro de investigación alemán de inteligencia artificial DFKI[53] ha realizado investigaciones en el área y desarrolló un intérprete de reglas de tipo encadenamiento hacia adelante utilizado para representar la interacción entre agentes de software denominado MAGSY [148].

Si bien inicialmente dichos trabajos parecen no tener una relación directa con el estudio que se realiza en este trabajo, es importante resaltar que la infraestructura que proporcionan los motores de evaluación de reglas permiten inferir nuevo conocimiento a partir de la base de reglas definida, siendo otra aplicación posible de dicha tecnología.

Otras universidades en donde se registra una importante cantidad de trabajos relacionados al área es la alemana Brandenburg University of Technology[54] promocionado principalmente por Adrian Giurca[55] y Gerd Wagner[56], en donde a través del proyecto REWERSE [77] se han realizado varios emprendimientos, entre ellos URML [176], una sintaxis basada en UML para especificar reglas de negocio, R2ML [223, 201], una especificación para el intercambio de reglas entre diferentes herramientas, Strelka [177], una herramienta para especificar reglas en forma visual integrando conceptos de ontologías y Web Semántica.

La Universidad Technische de Munich[57] principalmente gracias al trabajo de Adrian Paschke[58] registra varios trabajos relacionados, en particular *Reactive Rules* y *Reaction RuleML* donde se describe un lenguaje basado en reglas para la ejecución de reglas contractuales denominado RBSLA (Rule Based Service Level Agreement language) [196, 197, 200], y un esquema de validación y verificación de reglas con el objetivo de ayudar a los desarrolladores a determinar la confiabilidad de los resultados producidos por el sistema que administra las reglas de negocio [199].

Se registran trabajos en conjunto con Jens Dietrich de la Universidad neozelandesa Massey[59] investigando principalmente sobre bases de conocimiento [198], procesamiento de reglas de negocio [199], lenguajes de reglas encontrándose una activa participación en proyectos como Take [99] donde se utiliza tecnología de scripting para evaluar reglas y Mandarax [51] que aplica razonamiento hacia atrás a partir de reglas especificadas en RuleML [86].

[53] http://www.dfki.de/web
[54] http://www.tu-cottbus.de/btu/en.html
[55] http://www.informatik.tu-cottbus.de/~agiurca/
[56] http://www.informatik.tu-cottbus.de/~gwagner/
[57] http://ibis.in.tum.de/
[58] http://ibis.in.tum.de/staff/paschke/index.htm
[59] http://www-ist.massey.ac.nz/

Otro uso relevante de la tecnología se registra en varios trabajos de la Universidad de Alicante[60]. En [153, 154] se personaliza la interfaz de usuario de una aplicación web; para ello se utiliza un motor de reglas para modelar la parte variable de la lógica de la aplicación que genera la interfaz. Las reglas se aplican a las vistas de presentación y de navegación, logrando el soporte de la personalización deseada en tiempo de ejecución. Resulta de particular interés el uso de los motores de reglas en la generación de diferentes interfaces de usuario para múltiples dispositivos.

En relación con el ciclo de vida de una aplicación que utiliza reglas de negocio, existen varios trabajos relacionados. La Universidad de Lund[61] en Suecia dicta un curso curricular sobre diseño de sistemas de reglas de negocio por el docente Odd Steen[62]. En [164] se relaciona a las reglas de negocio con el comercio electrónico, en particular las relaciones B2C (business to consumer), en [190] se analiza el uso de reglas de negocio en el contexto de empresas suecas y en [112] se estudia el ciclo de desarrollo de software al incorporar reglas de negocio.

A su vez, en la Universidad Eslovena de Ljubljana[63], dentro de sus áreas de investigación, se han hecho estudios[64] sobre el enfoque de reglas de negocio, en particular a través varios trabajos, entre ellos los realizados por Marko Bajec[65] [120, 119] analizando principalmente metodologías y soporte de herramientas para la administración de reglas de negocio. Dichos trabajos se toman como referencia para el estudio del estado del arte en relación con el ciclo de desarrollo de software al incorporar el soporte de reglas de negocio.

Existen varios trabajos relacionados que aplican un enfoque similar al estudio que se realiza en este trabajo. La Universidad UCD de Irlanda[66] en [131] propone el uso de diferentes frameworks y metodologías para lograr aplicaciones que administren eficientemente conocimiento formal e informal utilizando varias herramientas open-source, entre ellas Drools para administrar las reglas de negocio.

En la Universidad Marquette de Wisconsin[67] en [173] explora el diseño de una aplicación centralizando el procesamiento de las reglas de negocio en forma independiente a la aplicación como mecanismo para responder en forma ágil a las condiciones cambiantes del negocio; utiliza el motor de reglas Clips [11].

Se registra un trabajo similar en la Universidad Technische de Hamburgo[68] donde también se explora el uso de reglas de negocio en un prototipo utilizando en este

[60] `http://www.dlsi.ua.es/iwad/`
[61] `http://www.ics.lu.se/en`
[62] `http://www.ics.lu.se/staff/odd.steen`
[63] `http://infolab.fri.uni-lj.si/`
[64] `http://infolab.fri.uni-lj.si/publications.html`
[65] `http://infolab.fri.uni-lj.si/marko/`
[66] `http://www.ucd.ie/`
[67] `http://www.marquette.edu/`
[68] `http://www.tu-harburg.de/`

caso herramientas de la empresa IBM ILOG en particular el motor de reglas para la plataforma Java, JRules.

En la Universidad Vilnius Gediminas de Lithuania[69] en [213] se analiza cómo las reglas de negocio se pueden utilizar para mejorar atributos cuantitativos y cualitativos en sistemas de información de tipo ERP (Enterprise Resource Planning[70]).

La Universidad alemana Karlsruhe[71] en [227] analiza y experimenta la problemática de lograr crear sistemas que utilicen reglas de negocio que sean fáciles de mantener y debuguear.

Resulta de particular interés el trabajo realizado en [136] en la Universidad Vrije de Bruselas[72]. Se propone una solución de administración de reglas de negocio independiente a la funcionalidad principal del sistema, incorporando elementos de la orientación a aspectos (AOP - Aspect Oriented Programming) [171] para conectar los conceptos de reglas de negocio con el código de la aplicación; también incorporando ideas de MDE (Model-Driven Engineering) [170] para lograr la especificación del sistema a un nivel de abstracción superior al que se obtiene con los lenguajes de propósito general y generar en forma automática las reglas de negocio y la aplicación que las utiliza. Se toma como referencia varias de las ideas del trabajo: por un lado el enfoque MDE tiene varios puntos de contacto con el desarrollo de software utilizando GeneXus [178] y por otro lado es interesante el planteo que realiza para transformar las reglas de negocio especificadas a un nivel de abstracción acorde al experto del negocio y lograr una representación ejecutable de las mismas.

Relacionado con los paradigmas de programación se registran varios trabajos. En la Universidad Addis Abeba de Etiopía[73] en [217] se realiza una comparación del enfoque de desarrollo de software basado en reglas BRA (Business Rules Approach) versus el desarrollo de software utilizando la orientación a objetos OOA (Object-Oriented Approach) resaltando fortalezas y debilidades. Se plantea un nuevo enfoque de desarrollo que integre lo mejor de ambos mundos denominado BROOM (Business Rules Object-Oriented Method).

En la Universidad Tecnológica Federal de Paraná[74] en [122] se propone un paradigma Orientado a Notificaciones donde combina el paradigma imperativo con el declarativo inspirándose en el manejo de hechos y reglas de los sistemas expertos.

En relación con la Web Semántica y ontologías se registran varios trabajos relacionados. La Universidad de Concordia en Montreal[75] en [127] investiga el uso de

[69] http://www.vgtu.lt/
[70] http://www.eccouncil.org/docs/ERP.pdf
[71] http://www.uni-karlsruhe.de/
[72] http://www.vub.ac.be/english/index.php
[73] http://www.aau.edu.et/
[74] http://www.utfpr.edu.br/
[75] http://www.concordia.ca/

un razonador basado en reglas utilizando lógica descriptiva. Utilizan la extensión SWRL [98] del lenguaje OWL y desarrolla un lenguaje de reglas con una sintaxis similar a SWRL pero con una semántica diferente. Implementa un razonador basado en reglas que extiende la definición de ontologías usando OWL denominado FIRE.

La Universidad Central de "Las Villas" en Cuba[76] detalla en [183] (trabajo presentado en el VII Congreso Internacional de Informática en Salud[77]) un modelo de hechos genérico que ayuda a representar las relaciones entre los conceptos de un dominio concreto haciendo uso de una ontología, implementándose una estructura genérica para el domino específico de Trasplante Renal.

En lo que refiere a la algoritmia que sustenta el área de estudio, existe una extensa bibliografía relacionada. Si bien no se realiza un estudio en profundidad de los diferentes algoritmos existentes, se detalla a continuación aquellos trabajos que reflejan los avances más importantes y se complementa en el Anexo D.

La Universidad Carnegie Mellon[78] a finales de la década de los 70 registra importantes aportes al área, en particular Charles Forgy [149, 150] presenta en su tesis de doctorado el desarrollo del algoritmo *Rete*, concepto fundamental utilizado o referenciado en la mayoría de los motores de evaluación de reglas. En [141] Robert B. Doorenbos publica su trabajo *Production Matching for Large Learning Systems* integrando el enfoque de sistemas basados en reglas con Machine Learning [182].

También la Universidad de Texas de Austin[79] tiene importantes aportes al área a lo largo de los años[80]; en particular se destaca la investigación realizada por Daniel Miranker creador del algoritmo TREAT [181].

Dentro del marco open source, el proyecto Drools registra en Drools Research Network Hub[81] el histórico de actividades académicas relacionadas con el proyecto, destacándose [214], para agregar la posibilidad de utilizar lógica difusa a la evaluación de reglas de negocio, [50, 214, 189, 188] desarrolla mecanismos para traducir reglas de negocio desde *Drools* [13] a *Jena* [40] y *Jess* [44] mediante un lenguaje de intercambio. En [192] se construye un sistema que es capaz de extraer reglas analizando sólo los datos creando un árbol de decisión que representa en forma compacta las reglas, utilizando algoritmos relacionados con Machine Learning [182].

Otros usos que si bien no tiene relación directa con el trabajo y resulta interesante resaltar es la evaluación de reglas en entornos de tiempo real y el procesamiento de eventos. La Universidad Politécnica de Valencia[82] en [208] realiza un análisis del uso de sistemas basados en reglas en entornos de tiempo real y propone un nuevo lenguaje

[76] http://www.uclv.edu.cu/
[77] http://www.informatica2009.sld.cu/
[78] http://www.cs.cmu.edu/
[79] http://www.cs.utexas.edu/
[80] http://www.cs.utexas.edu/research/publications/
[81] http://www.jboss.org/drools/research-network.html
[82] http://www.upv.es/

denominado Arlips basado en Clips que cumple con las condiciones necesarias de análisis temporal y de esa manera poder razonar y proporcionar una respuesta según las restricciones temporales que debe satisfacer un sistema de tiempo real.

La Universidad de Munich[83], en colaboración con IBM ILOG, investiga sobre la aplicación de reglas reactivas en la red [126], tecnologías relacionadas al procesamiento de eventos (CEP - Complex Event Processing) [175], es decir, la habilidad de detectar y responder a eventos en tiempo y forma. Es posible utilizar un motor de reglas para detectar, relacionar y seleccionar aquellos eventos relevantes y ejecutar las acciones que correspondan.

2.14. Conferencias

Existen diferentes tipos de conferencias relacionadas al área de estudio que se realizan periódicamente a lo largo del año; se ofrece una breve reseña de las mismas.

2.14.1. Rules Technology Summit

The Rules Technology Summit [89] expone a través de expertos cómo utilizar las diferentes tecnologías de reglas de negocio, que beneficios se obtienen y cómo se están utilizando en la industria.

2.14.2. Rules Expo

RulesExpo [88] es una conferencia en donde se exponen los diferentes productos existentes en el mercado cubriendo temas como reglas de negocio, administración de decisiones, procesamiento de eventos y tecnologías relacionadas.

2.14.3. EDM Summit

En Enterprise Decision Management Summit [15] se pretende obtener un método sistemático que permita automatizar y mejorar el proceso de toma de decisiones de la empresa [218]. Para esto los motores de evaluación de reglas de negocio son una herramienta más que se integran con BPM [139], Business Intelligence [185], arquitecturas SOA [144, 134] así como también integración de aplicaciones legadas para lograr el objetivo deseado.

2.14.4. Business Rules Forum

BRF [8] es el foro internacional de reglas de negocio; es una de las conferencias dedicadas a reglas de negocio más importantes que tiene como objetivo brindar herramientas y metodologías para maximizar la agilidad en el negocio.

[83] `http://www.pms.ifi.lmu.de/`

2.14.5. October RulesFest

October RulesFest [58] es una conferencia específica sobre motores de ejecución de reglas surgida en el 2008. En sus dos ediciones a la fecha (Marzo 2010) se ha logrado juntar a grandes investigadores en lo que refiere al tema de estudio, como por ejemplo Charles Forgy, creador del algoritmo Rete, David Luckham creador de CEP, Gary Riley creador de Clips y los principales exponentes de las herramientas líderes como por ejemplo JRules y Rules for Net, Blaze Advisor, Drools y Open Rules.

2.14.6. ISWC

International Semantic Web Conference [33] es el evento internacional más importante sobre la Web Semántica, en donde año a año se presentan aspectos de la web semántica: estado de arte, visión y adopción de la tecnología entre otros.

2.14.7. AAAI

La asociación para el avance de la inteligencia artificial [1] se fundó en 1979; es una sociedad científica sin fines de lucro abocada al avance del entendimiento científico acerca de los mecanismos subyacentes que sustentan el pensamiento y comportamiento inteligente y su respectivo uso en máquinas. Relacionado al área de estudio generalmente en estas conferencias, se presentan mejoras y avances en los algoritmos de pattern matching, infraestructura que sustenta a los motores de evaluación de reglas.

2.14.8. RuleML

The Rule Markup Initiative [86] tiene como objetivo intercambiar experiencia en el uso de tecnología de reglas, Web Semántica, sistemas multiagente, arquitecturas orientadas a eventos y servicios.

2.14.9. VORTE

Vocabularies, Ontologies and Rules for The Enterprise (VORTE [106]) es una serie de workshops que tienen el objetivo de intercambiar experiencias entre investigadores y la industria en áreas tales como sistemas de información, reglas de negocio, ingeniería basada en modelos y Web Semántica.

2.15. Organizaciones

Existen diferentes organizaciones especializadas en el uso de la tecnología de reglas de negocio y relacionadas. Una breve reseña se realiza a continuación.

2.15.1. BPMInstitute

BMPInstitute [4] es una organización sin fines de lucro creada para el intercambio de conocimiento relacionado a la administración de procesos de negocio, así como también proveer información sobre avances y desarrollos de ésta tecnología.

2.15.2. Business Rule Group

Business Rule Group [9] es una organización independiente dedicada a la adopción y práctica de metodologías asociadas con reglas de negocio.

2.15.3. BR Community

Business Rules Community [6] es una comunidad vertical no-comercial de profesionales relacionados al área de administración y gerenciamiento de reglas de negocio.

2.15.4. REWERSE

REWERSE [77] es una red de investigación relacionada con el "razonamiento en la web", con foco en la Web Semántica.

2.15.5. ONTORULE

ONTORULE [60] es un proyecto de la Unión Europea en donde se investigan tecnologías relacionadas con la Web Semántica y reglas de negocio. Uno de los intereses principales del proyecto es combinar ontologías, que se describen utilizando la lógica descriptiva (DL) con reglas de producción y reglas lógicas.

2.15.6. Business Rule Experts Group

BRXG [5] es una red social que nuclea a profesionales que utilizan tecnologías de reglas de negocio, forma parte de la Agility Alliance [2].

3. Propuesta

En este capítulo se describe la propuesta para la plataforma GeneXus [178]. En las secciones 3.1 a 3.4 se describen las principales características de la plataforma GeneXus y compara con el enfoque de desarrollo de software basado en modelos [191]. A partir de la sección 3.5 se detalla el desarrollo de la propuesta en sí.

3.1. Introducción

Luego de haber realizado un estudio de los diferentes enfoques de administración y uso de reglas de negocio y un estado del arte en estos temas, podemos decir que los elementos principales sobre los cuales se basa dicho enfoque son los términos que expresan los diferentes conceptos del negocio, los hechos que declaran verdades sobre dichos conceptos, y las reglas que definen restricciones sobre los elementos anteriores.

El enfoque de administrar las reglas de negocio en forma independiente a las aplicaciones introduce las reglas de negocio en dos etapas. En la primera, se presenta los elementos que constituyen el conocimiento del negocio (términos y hechos). Este conocimiento es la base sobre la cual se construye las reglas de negocio. En la segunda etapa, se expone las reglas de negocio en sí, definiéndolas y presentando una taxonomía de las mismas.

Antes de presentar los conceptos básicos y la propuesta de administración de reglas de negocio, se presenta una breve reseña de la herramienta GeneXus [178]. En particular se compara con el enfoque MDE [170] en el contexto de desarrollo de aplicaciones y posteriormente se detalla la propuesta de administración de reglas de negocio.

3.2. Descripción general de GeneXus

GeneXus [178] es una herramienta de especificación de sistemas de información basada en la aplicación de un modelo matemático (basado en los trabajos de Codd [137] y Warnier-Orr [193]) que permite capturar e integrar las visiones de los usuarios en bases de conocimiento (KB – Knowledge Base[1]) mediante objetos GeneXus y a partir de los cuales genera, mediante ingeniería inversa y procesos de inferencia, bases de datos normalizadas y aplicaciones completas [114]. Todo esto es generado para

[1] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?1836`

diferentes plataformas destino a partir de una misma especificación básica, pudiendo mantenerlas en forma automatizada ante cambios en los requerimientos, como detalla la Figura 3.1.

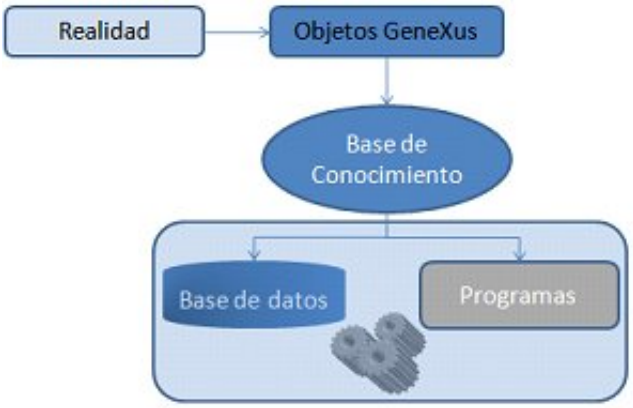


Figura 3.1.: Generación de programas y Base de Datos

El desarrollador GeneXus cuenta con distintos artefactos para crear una aplicación, los cuales se denominan objetos GeneXus[2].

3.2.1. Objetos GeneXus

Los principales objetos GeneXus se detallan en la Figura 3.2.

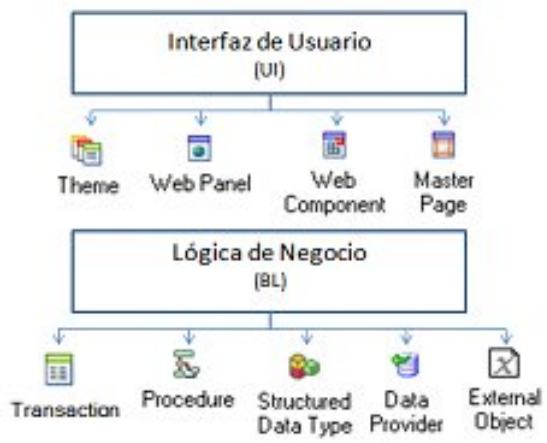


Figura 3.2.: Principales objetos GeneXus

Las Transacciones son utilizadas para definir la estructura de los datos, su forma básica de ingreso (interfaz gráfica) y las reglas de negocio que se desean aplicar. Detallaremos más sobre este tema en la sección 3.4.

[2] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?1866`

Los objetos Theme, WebPanel, WebComponent y MasterPage se utilizan para definir la interfaz gráfica.

Los objetos Procedure y DataProvider[3] contienen lógica pura, el primero de forma procedural y el segundo en forma declarativa.

Los objetos estructurados (SDT - Structured Data Type) representan, en la plataforma GeneXus, estructuras complejas.

Los objetos externos (External Objects) permiten la interoperabilidad de las aplicaciones GeneXus con el mundo exterior a través del consumo de Web Services, acceso a base de datos remotas utilizando procedimientos almacenados y uso de bibliotecas de terceros, como por ejemplo uso de DLL en C#, archivos JAR para Java y gemas para Ruby.

3.2.2. Modelando la realidad

El modelado de la realidad en el contexto GeneXus se consigue creando diferentes objetos (detallados anteriormente) que se almacenan en una Base de Conocimiento.

Para desarrollar la aplicación, GeneXus utiliza la información almacenada en la Base de Conocimiento, convierte este conocimiento en un metalenguaje propio para el cual realiza todas las verificaciones de integridad y consistencia; a este proceso se lo denomina *especificación* y genera un archivo interno con la representación correspondiente.

Posteriormente, tomando como entrada el resultado de la especificación, se genera código para el lenguaje configurado; a este proceso se lo denomina *generación*. En caso de existir cambios a nivel de estructura de datos se generan los programas para reorganizar las estructuras físicas correspondientes. Finalmente, dependiendo de la plataforma seleccionada se ejecutará el proceso de compilación y empaquetado de la aplicación.

Está basado en la especificación del modelado de datos conceptual y utiliza el supuesto de nombres únicos (URA - Universal Relation Assumption[4]), es decir, un mismo concepto se llama igual en todos los lugares donde se utiliza.

En la figura 3.3 se resumen los procesos principales que ocurren en la plataforma GeneXus, desde el modelado hasta la ejecución de una aplicación.

[3] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?5270`
[4] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?1870`

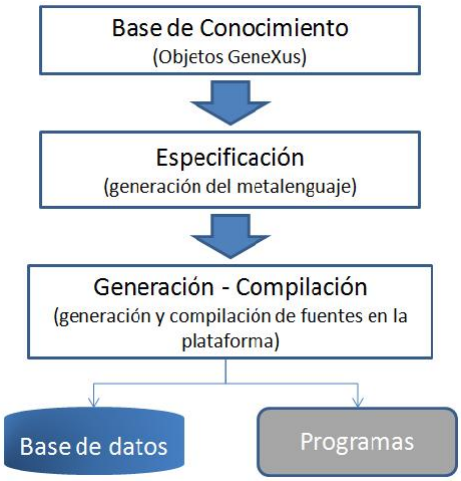


Figura 3.3.: Procesos en la plataforma GeneXus

3.2.3. Reseña Arquitectónica

A partir de la versión X, GeneXus posee una arquitectura extensible, por lo cual pueden agregarse al producto paquetes desarrollados por terceros para agregar y extender funcionalidad, conocidos como extensiones. La arquitectura se divide en 3 capas: User Interface, Business Logic y Data Layer.

La capa Business Logic está encargada de manejar la estructura básica, Base de Conocimiento, objetos GeneXus, sus partes, etc.

La extensibilidad de este diseño fue lograda mediante un esquema de paquetes y servicios:

- *Servicios UI*: KB, Command, Menu, ToolWindows, Documentos, Editores, Propiedades, Selección, Drag&Drop, etc.
- *Servicios GX UI*: Selección Att/Var, Objetos, Imágenes; Armado, Inspectors.
- *Servicios Data Layer*: CopyModel.
- *Servicios Business Logic*: KnowledgeManager, Search/Index.
- *Servicios GX BL*: Normalización, Ejecución, relación Tablas versus Transacciones, UserControlManager.

3.3. Desarrollo basado en Modelos

El desarrollo basado en GeneXus se trata de un enfoque en donde se independiza de la tecnología trabajando a un nivel de abstracción mayor que los lenguajes de propósito general, que se le denomina desarrollo basado en conocimiento.

Un enfoque similar existente en el mercado, es lo que se denomina MDE (Model Driven Engineering) [170] o MDA (Model Driven Architecture) [191] definido por la OMG [57], donde el desarrollo es conducido por modelos. Surge como una alternativa de desarrollo cuyo principal objetivo consiste en utilizar modelos de alto nivel de abstracción en las etapas de implementación, integración, mantenimiento y prueba de sistemas de software. La premisa de este paradigma es que el esfuerzo de desarrollo y mantenimiento del software está orientado hacia el trabajo a nivel de modelos y no a nivel del código fuente.

Se capturan las necesidades del negocio en un modelo y posteriormente, mediante diferentes mecanismos, se genera toda la aplicación para la plataforma deseada, logrando una programación a nivel del negocio totalmente independiente de la plataforma.

Se basa en la distinción fundamental entre 3 modelos diferentes [138]:

- Modelo Independiente del Cálculo o Computation Independent Model (CIM).
- Modelado lógico independiente de la plataforma, Platform-Independent Model (PIM).
- Nivel de implementación, Platform Specific Model (PSM), modelo específico para la plataforma.

3.3.1. CIM

El modelo independiente de cálculo se enfoca en el contexto y requerimientos del sistema sin tener en consideración su estructura o procesamiento. También se lo conoce como modelo de negocio o dominio porque utiliza el vocabulario que es familiar al experto del negocio.

Detalla exactamente qué es lo que tiene que hacer el sistema, ocultando toda la información tecnológica dejando fuera el *cómo* está construido o implementado el sistema. Este modelo juega un rol importante en acercar el mundo informático con los expertos del negocio.

3.3.2. PIM

El modelo PIM es una representación de la funcionalidad y comportamiento del negocio sin tener en cuenta detalles tecnológicos, a un nivel de abstracción menor que el modelo CIM.

3.3.3. PSM

Para implementar un modelo PIM en una plataforma específica, se necesita una herramienta que genere el modelo PSM a partir del PIM.

Dicha herramienta tiene que conocer la tecnología subyacente y saber realizar las traducciones\transformaciones correspondientes para lograr una representación ejecutable del modelo PIM.

El modelo PSM también es un modelo, pero a un nivel de abstracción menor, reflejando las características de la plataforma destino.

3.3.4. Transformaciones entre Modelos

Los dos conceptos principales de este enfoque son modelos y transformaciones. El patrón general se detalla en la Figura 3.4.

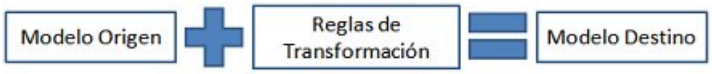


Figura 3.4.: Reglas de transformación entre modelos

Este enfoque tiene muchos puntos en común con lo que es la filosofía GeneXus en donde la representación del conocimiento se realiza en un modelo independiente de la plataforma, denominado Base de Conocimiento (conceptualizado como PIM en el contexto de modelado MDA), logrando, entre otras características, separar a los sistemas de información de la tecnología en la que ejecutan obteniendo una independencia tecnológica.

De esta manera se asegura que la inversión que se realiza en la construcción de los sistemas se preserva incluso cuando la tecnología subyacente cambia.

El diagrama de la Figura 3.5 detalla los grandes bloques constructivos del paradigma MDE [170].

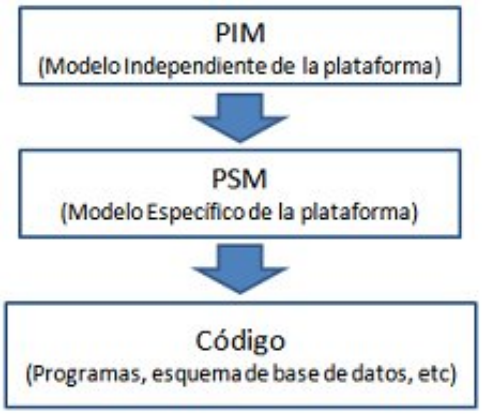


Figura 3.5.: Transformación de modelos a código fuente

Podemos asociar el modelo PIM a la especificación en GeneXus a través de los diferentes objetos GeneXus, como por ejemplo objetos *Transaction*, *Web Panel*, *Procedure*, *Data Provider* etc.

La representación del modelo PSM en la plataforma GeneXus no queda definida en forma explícita en el sentido de que no existe un modelo con dicha representación. Son los diferentes generadores GeneXus los componentes que representan dicho modelo, en donde se especifican en términos de propiedades características de la plataforma. Los generadores tienen el conocimiento para interpretar los objetos GeneXus a traves del metalenguaje resultado de la especificación (detallado en la sección 3.2.2), y según la configuración realizada, generar el código de la aplicación, esquema de la base de datos, etc. Es decir, el generador conoce la tecnología subyacente y realiza las traducciones\transformaciones correspondientes para lograr una representación ejecutable del modelo.

Por ejemplo, un objeto de negocio de tipo Transacción (modelo PIM) se traducirá a varios objetos del modelo PSM:

- Una definición de tablas usando SQL según el DBMS seleccionado.
- Los programas que crean la estructura de datos previamente definida.
- Una interfaz de usuario para insertar, actualizar y eliminar registros según la definición de tablas previamente definida.

3.4. Reglas en GeneXus

Como ya se mencionó en el capítulo anterior, las políticas y reglas empresariales se construyen a partir de componentes que forman el conocimiento básico del negocio.

Estos componentes son conceptos a los que llamaremos *términos*, y relaciones que llamaremos *hechos*. Para poder definir reglas, primero debemos definir los términos y hechos sobre los cuales se construirán las reglas.

En el contexto GeneXus, cuando se trabaja en el entorno de desarrollo, estamos definiendo reglas de negocios de muchas maneras, aunque no de la forma que utiliza el enfoque BRA [204].

Cuando defino un "value range" (para indicar que solo se permite un conjunto restringido de datos para un atributo o variable[5]); al definir elementos de tipo fecha en donde se realizan validaciones en forma automática; cuando se define un tipo de dato; un atributo que por medio del concepto URA (Universal Relation Assumption) que entre otras cosas define la estructura de la base de datos y las restricciones a nivel de Integridad referencial; también al definir formulas, cálculos, expresiones, etc, en todos estos casos estamos definiendo reglas de negocio.

Como se mencionó anteriormente, el objeto *Transaction* es utilizado para capturar no solo la estructura de los datos y definir su forma básica de ingreso (interfaz gráfica), sino también las reglas de negocio que se desean aplicar utilizando un lenguaje declarativo.

La definición de reglas tiene la siguiente sintaxis[6]:

```
Rule [ if condition ] [ on event... ] [ level att... ] ;
```

donde:

- `Rule` especifica un tipo de regla GeneXus.
- `condition` es una expresión booleana que detalla las condiciones que se deben cumplir para que la regla se ejecute. Puede componerse de operadores lógicos (`and`, `or`, `not`).
- `event` detalla la lista de eventos donde puede aplicar; los valores posibles son:

```
AfterInsert    |  AfterUpdate    |  AfterDelete
AfterValidate  |  AfterComplete  |  AfterLevel
BeforeValidate |  BeforeInsert   |  BeforeUpdate
BeforeDelete   |  BeforeComplete
```

- `att` es un atributo o lista de atributos separados por coma para determinar el nivel de la transacción donde debe de aplicarse la regla[7].

Una breve descripción de las principales reglas se detalla a continuación.

[5] http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?6797
[6] http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?6868
[7] http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?8213

3.4.1. Add/Substract

Si la condición definida por `condition` es verdadera, suma o resta el atributo `att1` en el `att2`.

```
Add( att1 , att2 )          [ if condition ] ;
Subtract( att1 , att2 )     [ if condition ] ;
```

3.4.2. Asignación

Si la condición definida por `condition` es verdadera, asigna el valor de la formula\expresión al atributo o variable.

```
att | &var = expression [ If condition ] ;
```

3.4.3. Call

Si la condición definida por `condition` es verdadera, ejecuta un objeto GeneXus.

```
Call( 'Usr-Pgm' , [ parm1 , parm2 , ... ] ) [ If condition ] ;
Pgm.Call( 'Usr-Pgm' , [ parm1 , parm2 , ... ] ) [ If condition ] ;
```

3.4.4. Submit

Si la condición definida por `condition` es verdadera, ejecuta en forma asincrónica un objeto GeneXus.

```
Submit( 'pgm' , 'parms' [ parm1 , ..., parmN ] ) [ If condition ] ;
```

3.4.5. Error/Message

Si la condición definida por `condition` es verdadera, en caso de utilizar la regla `Error` muestra un mensaje de error y finaliza la ejecución del objeto, mientras que al utilizarse con la regla `Msg` simplemente envía una notificación.

```
Error('msg' | &var | character_expression) If condition ;
Msg({'message' | &Var | character_expression}) [ If condition ] ;
```

3.4.6. Default

Asigna un valor predeterminado a un atributo o variable cuando se está realizando una inicialización.

```
Default( att | &var , expression );
```

3.4.7. Equal

Asigna un valor a un atributo en modo inserción; en los demás modos de ejecución se utiliza como filtro.

```
Equal( att , expression );
```

3.4.8. Serial

Genera un número serial.

```
Serial( att1 , att2 , step );
```

A partir del valor `att2` asigna el próximo serial teniendo en cuenta el incremento (step) y lo asigna en `att1`.

3.4.9. Consideraciones

Las reglas NoAccept[8], Accept[9], NoPrompt[10], Order[11], Prompt[12] no se toman en cuenta debido a que tienen que ver con la interfaz de usuario.

La regla Parm[13] define la firma del objeto; tampoco se toman en cuenta las reglas RefCall[14], RefMsg[15] específicas para realizar operaciones ante un error de integridad referencial al insertar un registro.

[8] http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?6844
[9] http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?6856
[10] http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?6861
[11] http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?8256
[12] http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?6863
[13] http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?6862
[14] http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?6864
[15] http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?6865

3.5. Propuesta BRA en GeneXus

En este trabajo, se propone administrar las reglas de negocio como un objeto en sí mismo dentro del entorno de desarrollo GeneXus, e implementar mecanismos que permitan que dichos objetos puedan llevarse al entorno de ejecución como metadata, de manera de poder variar dichas definiciones cuando sea necesario en el ambiente de ejecución.

3.5.1. Enfoque BRA

El enfoque BRA (Business Rule Approach) [204] y el *Manifesto de reglas de negocio* (ver Anexo A) plantean, entre otras cosas, lo siguiente:

1. Las Reglas son Independientes y no se encuentran contenidos en los procesos.
2. Declarativas, no procedurales.
3. Independiente de la tecnología.
4. Al servicio del negocio, no al de la tecnología.

Para el punto 1) es necesario extraer las reglas de negocio y administrarlas en forma externa a las aplicaciones ya que las reglas son un elemento esencial para los modelos de negocio. Debemos tratarlo como un "*ciudadano de primera clase*" en nuestro modelado.

Para el punto 2) es necesario poder expresarlas de forma declarativa en sentencias de lenguaje natural, expresado por el personal con conocimiento de negocio.

Para el punto 3) podemos decir que a largo plazo, las reglas son más importantes para el negocio que las plataformas de Hardware\Software donde se implementen. Las reglas de negocio son un patrimonio vital del negocio, por lo que administrarlas en forma independiente de la tecnología resulta fundamental.

Para el punto 4) implica que las reglas tienen que ser compartidas y comprendidas por toda la organización; como bien fundamental del negocio que son, no pueden quedar ocultas. A su vez, se deben de brindar herramientas que ayuden a formular, validar y gestionar las mismas.

3.5.2. Funcionalidad GeneXus

Como se mencionó anteriormente, el concepto de reglas de negocio ya existe a nivel de desarrollo GeneXus, pero las mismas no cumplen con muchas de las características deseables que propone el enfoque BRA [204].

Las reglas no son globales a la base de conocimiento; las mismas existen dentro del contexto de los objetos Transacción; no tienen la visibilidad deseada.

Si bien se definen en forma declarativa, no cumplen con el requisito de ser entendidas y comprendidas por toda la organización dado que tiene una semántica específica acorde al nivel de abstracción que maneja un desarrollador GeneXus.

Podemos afirmar que se satisface la característica de administración en forma independiente de la tecnología dado que la sintaxis no depende de ninguna implementación en particular ni lenguaje de programación, aunque su definición sintáctica sigue siendo de bajo nivel. A partir de dicha definición independiente de la plataforma (PIM), será el generador GeneXus utilizado el encargado de traducir dicha regla a la plataforma seleccionada (PSM).

3.5.3. Juego de Reglas

Uno de los objetivos propuestos consiste en "escalar" el concepto de regla a nivel de Base de Conocimiento. Es decir, que una regla (ver sección 3.4) sea un *ciudadano de primera clase* en el desarrollo GeneXus, teniendo un carácter global, manteniendo la definición declarativa.

A su vez, el concepto unitario de regla se agrupa en juego de reglas que denominaremos "*Ruleset*". Tomamos como referencia la sintaxis de definición de reglas de negocio GeneXus:

```
Rule [ if condition ] [ on event... ] [ level att... ] ;
```

Se realizan algunas modificaciones dado que las reglas serán un objeto en sí mismo, se pasa al siguiente esquema de definición:

```
Rule <nombre>
Propiedades
When
  <<condiciones>>
Then
  <<Acciones>>
```

donde la regla tendrá un nombre único que la identifique en la Base de Conocimiento, un conjunto de propiedades y dos grandes bloques constructivos: "*Condiciones*" y "*Acciones*" donde se aplicarán las acciones correspondientes si se cumplen las condiciones.

3.5.4. BOM, Hechos y Términos

Como se describió anteriormente en la sección 2.5, toda regla se basa en hechos, y los hechos en términos. La determinación del dominio se realiza identificando los términos utilizados en el negocio.

En muchos casos es importante desarrollar un glosario de términos del negocio para referenciar de forma no ambigua los diferentes elementos que conforman el

dominio de la aplicación. Varias herramientas utilizan el concepto de BOM (Business Object Model), un vocabulario independiente de la plataforma para representar el modelo de objetos de negocio en un lenguaje de negocio común. En el contexto de administración de reglas de negocio, este vocabulario contendrá la lista completa de términos para escribir reglas.

Para que el modelo de objetos del negocio (BOM) sea entendible por los analistas de negocio, es importante realizar una verbalización y adjuntar los términos y frases relacionados.

Supongamos un escenario típico de facturación donde se registran las siguientes entidades: Cliente (*Customer*), Producto (*Product*) y Factura (*Invoice*) y las siguientes operaciones:

- Registro de Clientes.
- Ingreso y categorización de Productos.
- Registro de Facturas.

La siguiente tabla verbaliza las propiedades más importantes de los objetos de negocio *Product*, *Customer* e *Invoice* que interesa para poder definir reglas de negocio:

Clase	**Miembros**	**Tipo**	**Verbalización**
Product	Msg("{msg}")	Void	Show Message "{msg}"
	&p.Productprice = &p.Productbase	Number	Assign Price
Customer	&c.CustomerTypeid = {value}	Number	Set Customer Type to '{value}'
	&c.CustomerMaxAllowed = {value}	Number	Set Customer Maximum Allowed to '{value}'
Invoice	RuleEngine.Halt()	Void	Halt Execution
	&il.InvoiceLineAmount = &il.InvoiceLineAmount - &il.InvoiceLineProductPrice	Number	Add a Free Product
	&i.Invoicediscount = &i.Invoicesubtotal * 0.05	Number	Accumulate Discount 0.05

En el contexto GeneXus, la principal fuente de identificación de hechos y términos son los objetos Transacción de tipo *Business Component*[16] (BC), objetos estructurados (SDT - Structured Data Types[17]), los atributos que los componen y sus relaciones, ya que entre otra cosas, dichos objetos conceptualizan los objetos del negocio, sus propiedades y valores.

La conexión lógica implícita entre las propiedades de cualquier objeto estructurado GeneXus (BC y SDT) determina la estructura que tendrán los hechos.

[16] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?5846`
[17] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?6286`

Tomemos como referencia el escenario de uso detallado anteriormente. La representación del Producto se centraliza en el objeto estructurado *Product*. Todos los atributos de dicho objeto (*ProductId*, *ProductName*, *ProductStock*, *ProductBase*, *ProductPrice*, *CategoryId*, *CityId* y *CountryId*) son considerados términos válidos del negocio que podrán utilizarse desde cualquier regla de negocio que se defina.

En la Figura 3.6 se plantean algunos ejemplos de ésta conceptualización que serán detallados en el capítulo 5 al analizar los escenarios de uso.

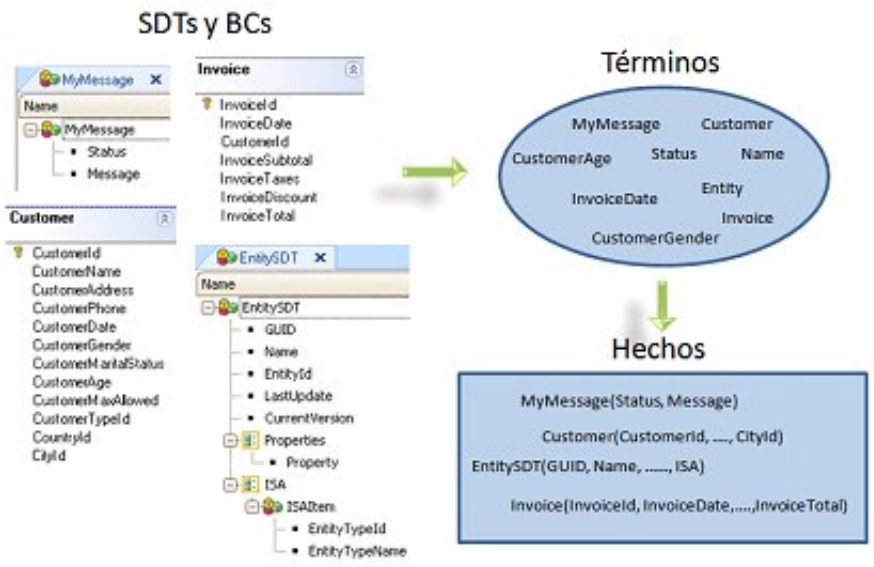


Figura 3.6.: Representación de términos y hechos

Los nombres de los términos no necesariamente deben coincidir con el nombre de los objetos GeneXus, atributos, etc; pueden denominarse de diferente manera para que tenga significado para el negocio, y justamente con el modelo BOM mediante la verbalización estamos definiendo un primer nivel de abstracción sobre la capa de implementación que representan los objetos GeneXus.

Categorización de Reglas

No existe un consenso en la clasificación de las reglas de negocio. Existen varias propuestas realizadas por diferentes empresas e individuos a lo largo de los años que por lo general se ajustan a diferentes implementaciones y propuestas comerciales [173].

Una clasificación de alto nivel de las reglas de negocio se puede dividir en tres grandes grupos mutuamente excluyentes: Restricciones, Producciones y Proyectores o Inferencia.

Para representar las reglas en el contexto GeneXus se decide utilizar esta clasificación.

Estructura de una Regla

Dividimos el problema de la representación de reglas de negocio en dos: por un lado, se plantea un lenguaje de reglas técnico que permita fácilmente utilizar los conceptos desde el ambiente de trabajo GeneXus. En una segunda etapa una vez definido este lenguaje, se plantea subir el nivel de abstracción de dicho lenguaje de manera de utilizar los términos y hechos en un lenguaje lo más "natural" posible para satisfacer el requerimiento: *las reglas deben ser compartidas y comprendidas por toda la organización*. Es decir, la representación de un lenguaje de reglas para la plataforma GeneXus tiene en cuenta dos dimensiones fundamentales:

- *Lenguaje técnico*: orientado al desarrollador GeneXus.
- *Lenguaje de "alto nivel" de abstracción*: orientado al analista del negocio.

Para la representación de reglas (formato técnico) se toma como referencia la representación PRR (Production Rule Repesentation) [71] donde se permite definir las reglas en dos grandes bloques constructivos: "*Condiciones*" y "*Acciones*".

Para la representación de reglas de negocio (formato alto nivel) se toma como referencia el estándar SBVR (Semantics of Business Vocabulary and Business Rules) [93].

La Figura 3.7 detalla los diferentes componentes de la solución propuesta para el tratamiento de reglas, relación con estándares y nivel de abstracción según el enfoque MDA [191].

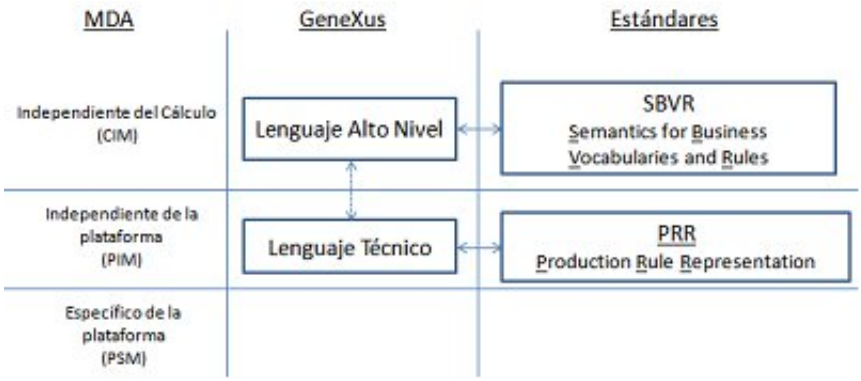


Figura 3.7.: Lenguajes de reglas para la plataforma GeneXus

En el modelo independiente de cálculo (CIM) se especifican las reglas de negocio al mayor nivel de abstracción utilizando el vocabulario definido que es familiar al

experto del negocio. Dicha representación tiene un mapping a un lenguaje de menor nivel de abstracción acorde al desarrollador GeneXus, manteniendo la independencia tecnológica (modelo PIM).

El nivel de abstracción específico de la plataforma constituye la representación en la propia plataforma de ejecución, el caso será analizado en el capítulo 4.

Reglas de Producción

Una regla de producción [71] es una sentencia de programación lógica que especifica la ejecución de una o más acciones en caso que una serie de condiciones se satisfaga. Se define sin tener en cuenta el orden de ejecución, dado que dicho orden se encuentra bajo el control del motor de inferencia permitiendo especificar múltiples condiciones y acciones.

En nuestro caso, cada regla tiene dos partes principales, denominadas *if-part* y *then-part*.

La sección *if-part* (también denominado condicional, antecedente) consiste de una lista de condiciones a ser testeada, en donde "si todas las condiciones de la sección son verdaderas", las acciones de la sección *then-part* (también denominada consecuente) serán ejecutadas.

La sintaxis planteada es la siguiente:

```
<conditions marker>
  <conditions>
<actions marker>
  <actions>
```

donde:

- <*conditions marker*> detalla que las expresiones a continuación deben de evaluarse por el motor de inferencia.

- <*conditions*>:
  - Una o más expresiones booleanas relacionadas por operadores lógicos (`and`, `or`, `not`).
  - Debe ser verdadero para que las acciones de la regla sean tenidas en cuenta para ejecutarse.

- <*actions marker*> indica que las siguientes expresiones son acciones (<*actions*>).

- <*actions*>:

- Una o más acciones.
- Se ejecutan en el orden que están definidas.
- Se requiere que las condiciones de la regla sean verdaderas antes de su ejecución.

Las reglas se interpretan de la siguiente forma: si las expresiones booleanas que se definen en las condiciones (<*conditions*>) son verdaderas, entonces las acciones (<*actions*>) se deben de ejecutar en forma secuencial.

Uno de los objetivos de modelar un espacio de decisión con reglas de producción es evitar la necesidad de especificar una secuencia en la ejecución de reglas. Es el motor de inferencia que administra la ejecución, disparo, y orden de las reglas basado en las condiciones que se cumplan en ejecución del programa y el estado interno del motor de evaluación.

Ejemplos

Para fijar ideas, supongamos la siguiente regla:

```
Clientes masculinos de hasta 25 años son de tipo A
```

En GeneXus quedará representada de la siguiente forma utilizando el lenguaje técnico de definición de reglas diseñado:

```
rule "Clientes masculinos de hasta 25 años"
  when
    &c:Customer(CustomerGender = "Male" , CustomerAge <= 25 )
  then
    &c.CustomerType = "A"
end
```

Las palabras clave `rule`, `when`, `then`, `end` definen la sintaxis básica de la regla de negocio utilizando el lenguaje técnico y se corresponde con los delimitadores de las secciones <*conditions marker*> y <*actions marker*>.

Las palabras clave `rule` y `end` definen el inicio y fin de una regla; `when` la representación del <*conditions marker*> que indica que a continuación se detallan un grupo de condiciones y `then` el <*actions marker*> que indica que a continuación se detallan las acciones de la regla.

Dentro de la sección <*conditions marker*> la siguiente expresión detalla las condiciones que se deben de satisfacer para que la regla se ejecute:

```
&c:Customer(CustomerGender = "Male", CustomerAge <= 25 )
```

`Customer` es representación GeneXus del cliente en el ambiente de desarrollo GeneXus. Deberá existir un objeto de tipo *Transacción* o *SDT* que lo represente con al menos las siguientes propiedades: `CustomerGender` y `CustomerAge,` que definen dos condiciones que restringen al cliente.

Las condiciones a aplicar sobre un objeto se encierran entre paréntesis y en caso de utilizar varias condiciones se conectan utilizando la coma "," o el operador `and`, en este caso:

```
(CustomerGender = "Male", CustomerAge <= 25 )
```

`&c` se corresponde con la proyección del cliente que satisface las condiciones en una variable que podrá utilizarse en las acciones de la regla.

La sección <*actions marker*> detalla la siguiente acción:

```
&c.CustomerType = "A"
```

A través de la proyección del cliente de la sección de condiciones utilizando la variable `&c` se accede a la propiedad `CustomerType` del cliente y se modifica su valor.

Esto nos permite definir un lenguaje declarativo liviano a nivel de puntuación, fácil de entender y usar para un desarrollador GeneXus.

Algunos ejemplos de cómo especificar reglas de tipo *Restricción*, *Producción* y *Proyección* (detallado en la sección 2.5.6) en formato técnico se detalla a continuación.

Regla Restricción Son reglas que devuelven un error si se verifica que se ha llegado a un estado inconsistente del sistema. Por ejemplo:

```
Un cliente tiene que tener un nombre.
```

Utilizando la notación elegida se puede definir de la siguiente manera:

```
rule "Rechazar Cliente sin nombre"
  when
    &c:Customer(CustomerName == "")
  then
    Msg("Ingrese nombre válido...")
    &c.Error = true
end
```

Se define una regla de tipo restricción de nombre "*Rechazar Cliente sin nombre*" que indica que el nombre del cliente es requerido.

Cuando la condición de la regla se cumpla (especifica que la propiedad `CustomerName` es vacía para el objeto `Customer`), en la sección de acciones se muestra un mensaje detallando la restricción y actualiza un atributo booleano `Error` informando que existe algún error en dicho objeto.

Regla Producción Son reglas que realizan o derivan un cálculo basado en alguna función matemática, de uso similar a las funciones en otros paradigmas.

Supongamos se define la siguiente regla:

```
Los productos de procedencia china que sean juguetes
aplicar un arancel del 20% sobre el precio base
```

Se podría representar como:

```
rule "Juguetes Chinos, aplicar un 20% de arancel"
  when
    &p : Product( Countryid == 5 , Productcategoryid == 2 )
  then
    Msg("Juguetes Chinos, aplicar un 20% de arancel")
    &p.Productprice = &p.Productprice + &p.Productbase * 0.2
end
```

Cuando las condiciones de la regla se cumplan (la propiedad `CountryId` es 5 y `ProductCategoryId` es 2 para el objeto `Product`), la acción:

```
&p.Productprice = &p.Productprice + &p.Productbase * 0.2
```

se encarga de calcular el precio del producto proyectado en la variable `&p` y almacenarlo en la propiedad `ProductPrice` a partir del precio base representado por la propiedad `ProductBase`. Además, utilizando la función `Msg`

```
Msg("Juguetes Chinos, aplicar un 20% de arancel")
```

se despliega un mensaje informando que la regla ha sido ejecutada.

Regla Proyección Son reglas que toman alguna acción cuando ocurre un evento relevante. Por ejemplo:

```
Si un cliente es de alto riesgo
notificar al centro de atención a clientes
```

Asumiendo que un cliente de alto riesgo es aquel que tiene más de 70 años, la regla de tipo proyección se podría definir de la siguiente forma:

```
rule "Informar a Atención al cliente los clientes de Alto riesgo"
  when
    &c : Customer( Customerage > 70 ) // de alto riesgo
  then
    Msg( "Clientes de alto riesgo, informando a Help Desk")
    UpdateCustomer.Call(&c,"Alto Riesgo")
End
```

Cuando la condición de la regla se cumpla (la propiedad `CustomerAge` supera el valor 70 para el objeto `Customer`), en las acciones de la regla se llama a un procedimiento GeneXus denominado `UpdateCustomer` que se encargará de realizar las operaciones necesarias para notificar al centro de atención de clientes el caso recibiendo como argumentos el cliente proyectado en las condiciones de la regla y una constante.

```
UpdateCustomer.Call(&c,"Alto Riesgo")
```

3.5.5. Lenguaje de Alto Nivel

El lenguaje de reglas definido anteriormente, si bien satisface varios de los puntos detallados en el enfoque BRA (Business Rules Approach) [204] (sección 3.5.1), no tiene el nivel de abstracción suficiente para estar *al "servicio del negocio"*. Es decir, las reglas solo son utilizables por un desarrollador GeneXus.

El objetivo de lograr un lenguaje de alto nivel de abstracción de reglas de negocio que sea fácilmente entendible por el experto del negocio, no se logra con el lenguaje técnico detallado en la sección anterior.

Para que la representación de reglas de negocio sean comprendidas y compartidas por toda la organización, se toma como referencia el estándar SBVR (Semantics of Business Vocabulary and Business Rules - Semántica de Vocabulario y Reglas de Negocios) [93] impulsado por la OMG, formalmente definido como una taxonomía que describe operaciones y reglas de negocio elementales.

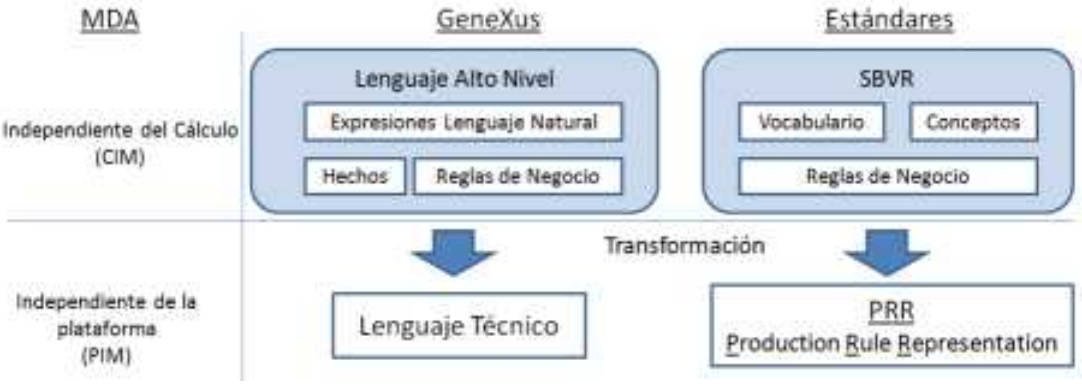


Figura 3.8.: Estándar SBVR versus Lenguajes de reglas "Alto nivel" GeneXus

Lo interesante de este estándar (detallado en la Figura 3.8) es que las reglas se especifican a través de un vocabulario que define las construcciones básicas del lenguaje y se logra una representación muy cercana al lenguaje natural.

Las reglas detalladas en la sección 3.5.4 en formato SBVR [93] se podrían definir de la siguiente forma:

```
Un cliente tiene que tener un nombre.

Los productos de procedencia china que sean juguetes
aplicar un arancel del 20% sobre el precio base.

Si un cliente es de alto riesgo
notificar al centro de atención a clientes.
```

donde:

- Texto subrayado se corresponde con el vocabulario, conceptos y términos de negocio.
- Texto en itálica detalla relaciones entre conceptos.

Notar que la representación SBVR [93] tiene como público objetivo los expertos de negocio. Se pretende lograr una definición de alto nivel de reglas que describa y represente el mundo en el cual el negocio opera expresado en un lenguaje natural estructurado. Es por eso que dentro del modelo MDA [191] se posiciona en la capa superior CIM (modelo independiente del cálculo).

Se deja de lado el procesamiento automático de las reglas a este nivel; sin embargo, la especificación en formato alto nivel se transforma a la representación formato técnico que sí tiene en cuenta su automatización en un ambiente de ejecución.

Es importante resaltar que soporta totalmente la definición inicial de reglas de negocio en donde las reglas se construyen a partir de hechos, y éstos se construyen a partir de términos. Se está agregando un nivel de abstracción superior para facilitar la comunicación con los expertos del negocio (detallado en la Figura 3.9).

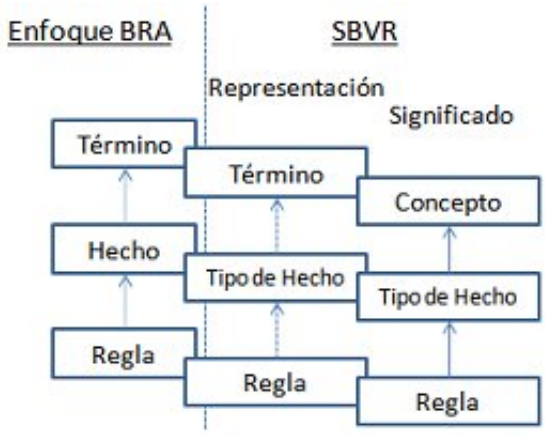


Figura 3.9.: Estándar SBVR versus enfoque BRA

Para la construcción de este lenguaje se toma como referencia el enfoque DSL (Domain Specific Language) [169], es decir se construye un lenguaje específico para el dominio de reglas de negocio. Un *DSL* se define como:

> *Un lenguaje de programación de expresividad limitada que se enfoca en un problema de dominio particular*

Un DSL es un lenguaje que se diseña para resolver un conjunto específico de tareas. Se focaliza en problemas dentro de un dominio particular sin pretender brindar soluciones para los problemas que están fuera de éste.

El objetivo en el contexto del trabajo, es lograr un lenguaje de reglas de negocio que extienda el lenguaje "técnico" para que sea fácilmente entendible por el experto del dominio, ocultando las particularidades del lenguaje técnico utilizado por el desarrollador GeneXus. Define un vocabulario de negocio que contiene todos los términos especializados y definiciones de conceptos que una organización utiliza para "hacer negocios".

Por ejemplo, supongamos que definimos un vocabulario independiente de la plataforma como detalla la Figura 3.10:

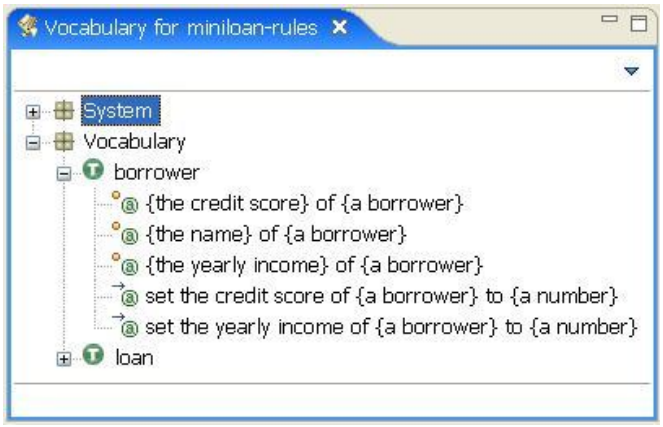


Figura 3.10.: Un vocabulario de ejemplo

A partir de dichas construcciones se pretende que la definición de reglas de negocio permita utilizar bloques de lenguaje natural como detallan las Figura 3.11 y Figura 3.12. Al utilizarse en la sección de condiciones de una regla:

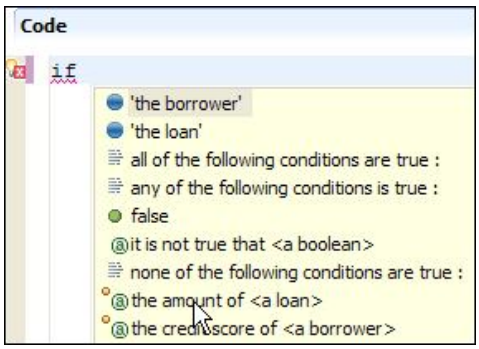


Figura 3.11.: Condiciones de una regla

En la sección de consecuencias o acciones de una regla:

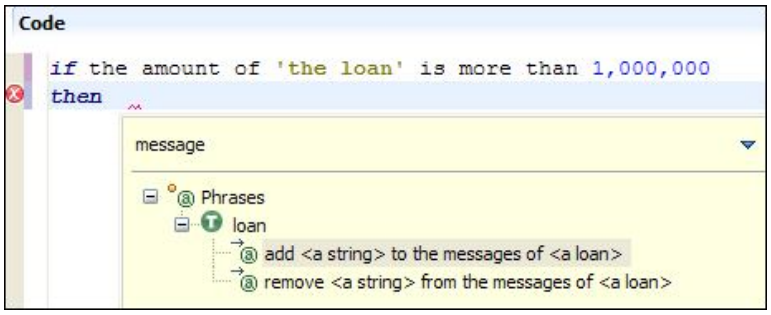


Figura 3.12.: Consecuencias de una regla

El implantador define el lenguaje de modelado que contenga a todos los conceptos que sean necesario para definir las reglas de negocio en los términos que el experto del negocio entiende. Es decir, será necesario verbalizar términos y hechos para poder construir reglas en base a dichos elementos. Posteriormente, el implantador es el encargado de especificar el mapping o traducción al lenguaje técnico de reglas GeneXus (definido a partir de la sección 3.5.3) utilizando los correspondientes objetos GeneXus de la Base de Conocimiento.

Se está utilizando el concepto de BOM (Business Object Model) definido anteriormente, un vocabulario independiente de la plataforma para representar el modelo de objetos de negocio en un lenguaje de negocio común. Es decir, en el contexto de administración de reglas de negocio contendrá la lista completa de términos para definir reglas.

También se representa el modelo de objetos ejecutable XOM (eXecutable Object Model) que es la información que manipula el personal técnico encargado del desarrollado de la aplicación.

Supongamos una definición de un lenguaje de alto nivel de abstracción de ejemplo que define un mapping al lenguaje técnico de reglas como se detalla a continuación.

Lenguaje de Alto Nivel	Lenguaje Técnico	Contexto
fin	End	palabra clave
regla	Rule	palabra clave
cuando	When	palabra clave
entonces	Then	palabra clave
El Producto	&p : Product( )	Condición
Does not have a name	CustomerName==""	Condición
Customer	&c:Customer()	Condición
Juguete	Productcategoryid == 2	Condición
Customer age is greater than '{age}'	&c : Customer(CustomerAge > {age})	Condición
De China	Countryid == 5	Condición
Customer Has an Error	&c.Error = true	Acción
Log "{message}"	Msg("{message}")	Acción
Mostrar Mensaje "{m}"	Msg("{m}")	Acción
Incremetar Producto Base en {b}	&p.Productprice = &p.Productprice + &p.Productbase * {b}	Acción
Update The Customer With "{t}"	UpdateCustomer.Call(&c,"{t}")	Acción

A partir de las expresiones del lenguaje natural (detalladas en la columna *Lenguaje de Alto Nivel*) se pueden obtener reglas de negocio fácilmente entendibles para el experto del negocio manteniendo la representación en formato técnico (columna *Lenguaje Técnico*) como detalla la Figura 3.13.

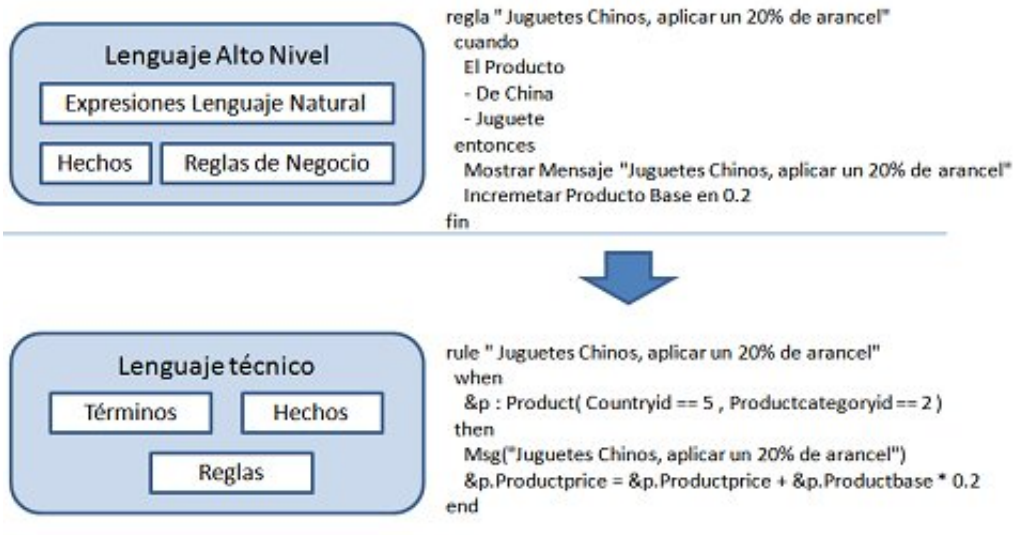


Figura 3.13.: Traducción de regla en formato alto nivel a técnico

Las reglas (formato técnico), detalladas anteriormente en la sección 3.5.4, se pueden reformular de la siguiente forma utilizando las expresiones del lenguaje de alto nivel de abstracción.

Regla de restricción utilizando un pseudo lenguaje inglés:

```
rule "Rechazar Cliente sin nombre"
  when
    Customer
     - Does not have a name
  then
    Log "Ingrese nombre válido..."
    Customer Has an Error
end
```

Regla de producción utilizando un pseudo lenguaje español:

```
regla "Juguetes Chinos, aplicar un 20% de arancel"
  cuando
    El Producto
     - De China
     - Juguete
  entonces
    Mostrar Mensaje "Juguetes Chinos, aplicar un 20% de arancel"
    Incremetar Producto Base en 0.2
fin
```

Regla de proyección utilizando un pseudo lenguaje inglés:

```
rule "Informar a Atención al cliente los clientes de Alto riesgo"
  when
    Customer age is greater than '70'
  then
    Log "Clientes de alto riesgo, informando a Help Desk"
    Update The Customer With "Alto Riesgo"
End
```

La esencia del lenguaje de reglas de "alto nivel" de abstracción que propone el trabajo es la siguiente: a partir de los constructores básicos que definen la estructura de una regla de negocio y un juego limitado de expresiones en un lenguaje arbitrario totalmente parametrizable, se construyen reglas a un nivel de abstracción mucho mayor logrando el objetivo planteado. Dado que dichas expresiones tienen una traducción directa al lenguaje técnico de reglas desarrollado, se obtiene una representación ejecutable de las reglas en la plataforma GeneXus.

3.5.6. Evaluador

El *Evaluador*, es el componente que permite ejecutar las reglas de negocio, sobre instancias de objetos del modelo de conceptos definido para un negocio en particular.

Un motor de reglas provee una alternativa que brinda capacidades de este estilo, en donde se declaran las reglas al nivel que el analista de negocio desea, y un motor

toma estas definiciones y a partir de los datos recibidos ejecuta lo que indiquen las reglas.

Para representar el motor de evaluación se crea un objeto externo GeneXus de tipo *External Object*[18] denominado *RulesetEngine*. Se define un conjunto de funcionalidad básica para poder evaluar reglas en tiempo de ejecución:

- Evaluar un juego de reglas utilizando la sección de condiciones.
- Insertar, modificar y eliminar objetos de la evaluación, modificando internamente la memoria de trabajo del motor de ejecución.
- Ejecutar la sección de acciones de una regla.

3.5.7. Mecanismo de Disparo

En las secciones anteriores se desarrollan los conceptos básicos para representar las reglas en un nuevo objeto GeneXus y se define un objeto de tipo *External Object* que representa el motor que las evalúa. Se necesita disponer de un mecanismo que permita relacionar ambos conceptos, es decir, especificar cuando se quiere aplicar un juego de reglas. Esto es lo que se especifica en la cláusula <*On*> en la sintaxis de definición de reglas en GeneXus (detallado en la sección 3.4).

En resumen, se necesita asignar uno o varios juegos de reglas al evaluador y posteriormente definir sobre cuáles objetos se pretende aplicar las reglas definidas. Para ello nos basamos en el patrón *ECA Rules* (Event Condition Action Rules) [118, 161] o "Production Rules" que define:

- *Evento*: especifica cuándo quiero evaluar las reglas.
- *Condición*: test lógico a evaluar.
- *Acción*: acciones a ser ejecutadas si la regla cumple las condiciones.

Las *condiciones* y *acciones* están autodefinidas en lo que son las reglas en sí; el evento de disparo se especifica en forma explícita por medio de código procedural indicando el juego de reglas a evaluar y el juego de datos correspondiente.

Supongamos que se quiere evaluar un juego de reglas antes de persistir un cliente. Un pseudo código que utilice el evaluador con un juego de reglas encapsulado en un objeto *"Ruleset"* sería:

```
VariableCliente = new()
VariableCliente.Nombre = "ejemplo"
// configurar información
VariableCliente.Edad = 18
```

[18] http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?5669

```
...
...
// juego de reglas a evaluar
Engine.Ruleset( "rulesetname")
Engine.EvaluarUn( VariableCliente )
Engine.DispararReglas()
```

La evaluación de las reglas de negocio implica la interacción entre la aplicación generada, el juego de reglas definido, y un motor de evaluación de reglas que se encargue de aplicar las acciones correspondientes cuando las reglas satisfacen sus condiciones de disparo.

Es decir, hay un cambio importante a nivel de arquitectura de la aplicación. Anteriormente, las reglas de negocio se encontraban ocultas y embebidas en la lógica de negocio de la aplicación en forma procedural (componente BL – Business Logic) como detalla la Figura 3.14:

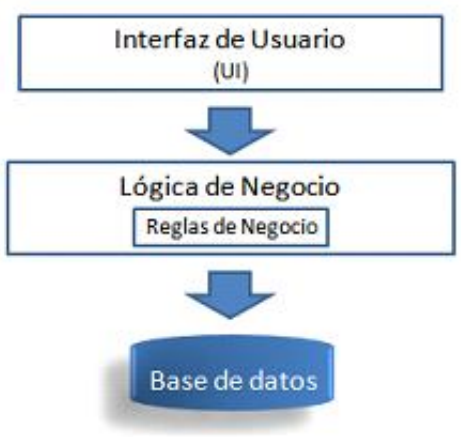


Figura 3.14.: Reglas de Negocio embebidas en la aplicación

La propuesta BRA [204] para la plataforma Genexus, implica que las reglas de negocio clave (representadas utilizando objetos *Ruleset*) se encuentran separadas de los demás objetos que se corresponden con la lógica de negocio. A su vez, se agregan nuevos componentes como es el evaluador de reglas (representado por un objeto externo) y un editor externo de reglas, que permitirá modificar y agregar nuevos juegos de reglas en tiempo de ejecución con el objetivo de lograr la flexibilidad planteada. La nueva arquitectura de la aplicación se detalla en la Figura 3.15:

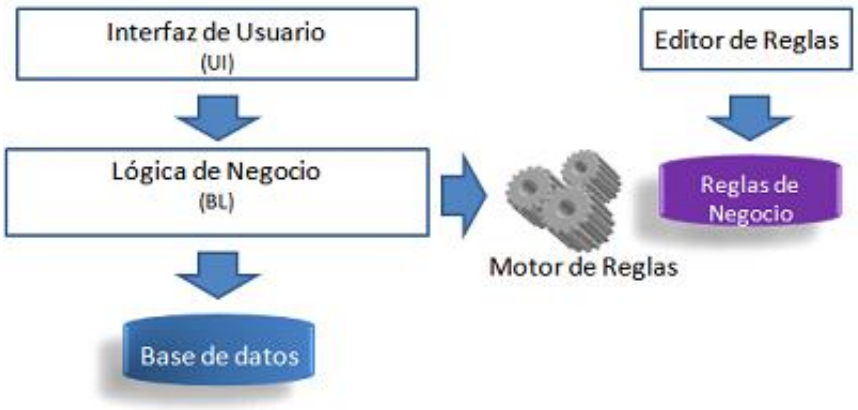


Figura 3.15.: Reglas de Negocio independientes de la aplicación

Como primer paso, en forma explícita en la codificación GeneXus, se indica proceduralmente que se desea evaluar un juego de reglas. Posteriormente, se le envía información de contexto y ejecuta la evaluación de reglas. Una vez finalizada la ejecución de reglas se continúa con la ejecución normal del programa generado.

Las reglas son independientes entre sí y nunca se ejecutan en forma explícita, sino que se ejecutan en forma implícita por el motor de ejecución cuando las condiciones de las mismas se satisfacen.

Es decir, todas las reglas son evaluadas dependiendo de los datos de contexto (memoria de trabajo) que existan una vez que se ejecuta el motor de evaluación. A su vez, por el mecanismo de inferencia y la existencia de operadores explícitos en la sección de acciones, como por ejemplo *insert*, *update* y *retract*, forzará a que se evalúen nuevamente las reglas en la medida que los hechos se modifiquen.

De esta forma se logra desarrollar aplicaciones más ágiles utilizando reglas de negocio fácilmente modificables en el contexto de ejecución.

3.6. Editor Externo de Reglas

Las reglas de negocio no son estáticas; en ciertas ocasiones se necesitan hacer modificaciones para permanecer actualizado con respecto a las demandas del mercado y expectativas de los clientes.

El editor externo de reglas es el módulo que permite administrar (agregar\modificar\eliminar) reglas de negocio en forma independiente del software que las utiliza en el ambiente de producción. Las modificaciones que se hagan sobre el juego de reglas impactan en la ejecución de la aplicación que las utiliza en forma instantánea.

Los programadores se encargan de crear las reglas iniciales e infraestructura base necesaria utilizando el entorno de desarrollo GeneXus o el editor externo de reglas de negocio. En caso de utilizar el entorno de desarrollo, las reglas e infraestructura base se exportan como archivos utilizando un formato XML y depositan en el repositorio de reglas. Los analistas de negocio tienen la posibilidad de editar las reglas en el entorno de ejecución, devolviendo el control de las políticas y restricciones del negocio al experto (ver Figura 3.16).

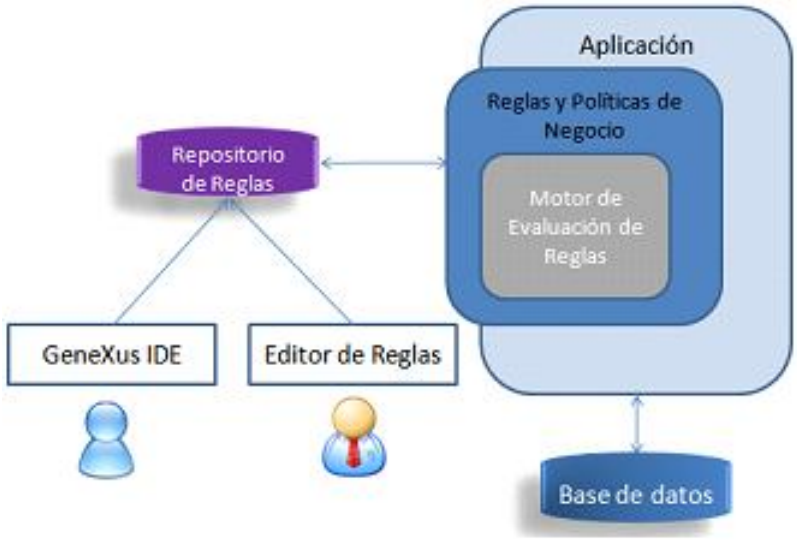


Figura 3.16.: Edición externa de reglas de negocio

Se administra un repositorio de reglas de negocio para ser accedido desde las aplicaciones y el diseñador de reglas de negocio utilizando el modelo de objetos del negocio (BOM), términos y hechos. La aplicación, además de interactuar con la base de datos, interactúa con el motor de evaluación de reglas que utiliza la base de reglas disponible.

El editor externo de reglas debe conocer:

- El modelo de objetos del negocio, que serán los objetos sobre los cuales se podrán definir y modificar reglas generalmente utilizado en la sección de condiciones de las reglas y proyectado como variable para ser referenciado desde las acciones. En el contexto GeneXus se necesita disponer del catálogo de objetos estructurados que componen la base de conocimiento en tiempo de ejecución.
- Qué acciones son factibles de ser ejecutadas sobre los objetos del negocio para utilizarse en la sección de consecuencias de las reglas; en el contexto GeneXus se corresponde con Procedimientos, DataProviders, métodos y propiedades disponibles para manipular objetos estructurados.

- El juego de reglas de negocio disponible y los lenguajes de alto nivel utilizados para definirlas.
- La plataforma y tecnología subyacente donde se ejecuta la aplicación para poder resolver las traducciones\transformaciones correspondientes a la plataforma.

Se debe administrar además el ciclo de vida de las reglas de negocio, así como también gestionar su evolución.

3.6.1. Ciclo de Vida

Cuando se modifica un juego de reglas se deberá utilizar el editor de reglas correspondiente para reflejar el cambio. Una vez que el juego de reglas se actualiza, se realizan las validaciones correspondientes utilizando los lenguajes de reglas disponibles, modelo de objetos y hechos, almacenando la nueva definición.

La próxima ejecución del juego de reglas tendrá en cuenta los recientes cambios realizados a la especificación de reglas.

3.6.2. Gestión de Cambios

Resulta muy importante identificar y clasificar los diferentes componentes del sistema teniendo en cuenta su afinidad al cambio. Es decir, no todos los componentes de una solución evolucionarán de la misma forma. En particular, muchas de las reglas de negocio que se definan tendrán una volatilidad mayor con respecto a otros artefactos en lo que al ciclo de vida de la aplicación refiere, incluso diferentes reglas de negocio tendrán diferente grado de volatilidad.

Se busca minimizar el tiempo necesario para actualizar el sistema facilitando la modificación de aquellos componentes que más lo necesitan. En general, se estima que el tiempo promedio sin que sea necesario modificar una aplicación sin reingeniería varía entre 3 a 5 años [156]. Para una aplicación que utilice en forma intensiva un motor de evaluación de reglas de negocio, extiende automáticamente la vida útil de la aplicación al menos en 2 años. Esto se debe a que las reglas de negocio NO son administradas como un requerimiento estándar, sino que es metadata de la propia aplicación, permitiendo que las mismas cambien sistemáticamente añadiendo un valor significativo para las empresas.

3.7. Beneficios del Enfoque

El enfoque BRA [204] estudiado resalta la importancia de administrar en forma independiente de las aplicaciones a aquellas restricciones y políticas del negocio que sean críticas y tengan alta volatilidad.

La propuesta en este trabajo sugiere administrar las reglas de negocio en forma declarativa como un objeto en sí mismo dentro del entorno de desarrollo GeneXus, desarrollando lenguajes de especificación de reglas en el contexto del desarrollador GeneXus y de alto nivel de abstracción con sentencias en lenguaje natural para ser utilizadas por el experto en el dominio.

En el ambiente de desarrollo GeneXus se continúa trabajando a nivel de conocimiento (equivalente PIM del enfoque MDA [191]), se agregan nuevos artefactos para poder utilizar las reglas de negocio de otra forma y lograr mayor flexibilidad y por lo tanto mejorar la agilidad de los procesos empresariales (ver Figura 3.17).

Se propone además, un lenguaje de definición de reglas personalizable a un nivel de abstracción mayor (nivel CIM en MDA) para que los expertos del negocio puedan entender y gestionar las reglas empresariales y en consecuencia influir en forma directa en la aplicación.

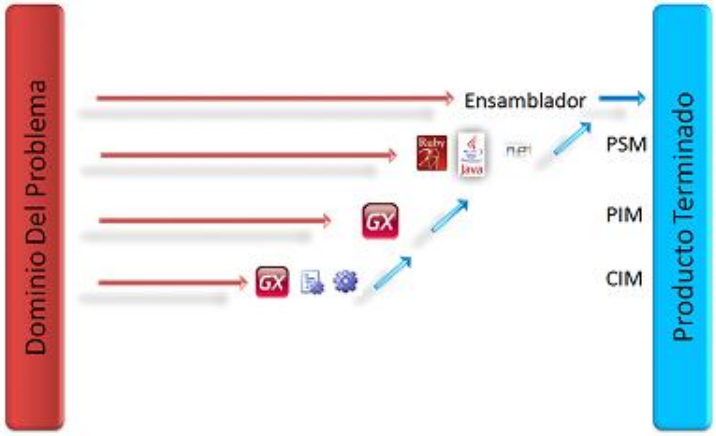


Figura 3.17.: Abstracción en el modelado

Las reglas de negocio se administran en forma independiente a la aplicación y disponibiliza un editor de reglas para ser utilizado en el entorno de ejecución con el objetivo de lograr flexibilidad al implantar y mantener aplicaciones.

4. Implementación

En este capítulo se describe la implementación realizada. En la sección 4.1 se resume la selección de herramientas realizada para las diferentes plataformas de ejecución GeneXus [178]. A partir de la sección 4.2 se describe la implementación según la propuesta definida en el capítulo 3.

4.1. Evaluación de Herramientas - ecosistema GeneXus

Utilizando el análisis de herramientas existentes en el mercado detallado en el capítulo 2, se analiza al menos una herramienta para las principales plataformas de ejecución GeneXus: Java, .Net y Ruby.

Por el detalle del criterio de evaluación utilizado y la selección de herramientas verificar el Anexo E.

En términos generales la selección y evaluación consistió en:

- *Pre-selección*: Seleccionar una lista de herramientas candidatas a partir del estudio de herramientas detallado en el capítulo 2 priorizando aquellas herramientas de código abierto.
- *Evaluación*: Se desarrollan los casos de uso que detalla el Anexo G utilizando código generado GeneXus.
- *Selección*: Se seleccionan aquellas herramientas que logren resolver los casos de uso planteados.

Para el ambiente *Java* es donde se registra la mayor cantidad de opciones open-souce. Después de analizar diferentes opciones, se decide revisar en más detalle las propuestas *Drools*, *Jena Rules* y *Jess*.

Para la plataforma *.Net* se pre-seleccionan las herramientas *NxBRE* y *Drools.NET*.

Para *Ruby* se selecciona la gema open source *Ruleby*.

Se resume a continuación la estrategia de evaluación, resultados obtenidos y selección realizada.

4.1.1. Plan de Evaluación

La estrategia de evaluación consistió de los siguientes pasos:

- Se realiza una pre-selección de herramientas para las principales plataformas de ejecución GeneXus: Java, .Net y Ruby teniendo en cuenta los criterios de evaluación que detalla el Anexo E.
- Para cada herramienta seleccionada, se instala el motor de reglas y estudia el juego de ejemplos incluidos en caso que exista. Esto incluye la especificación del modelo de objetos, reglas y hechos, y la interacción desde el lenguaje procedural con el motor correspondiente.
- Se realiza la primera prueba de concepto, se implementa el caso de uso "*Hola Mundo*" (detallado en el Anexo G) para asegurarse que las funcionalidades básicas son soportadas por el motor. En caso que la evaluación sea satisfactoria, se sigue adelante con los escenarios de uso relacionados al ejemplo GeneXus; de lo contrario, se selecciona el siguiente motor a evaluar.
- Se genera el escenario de prueba GeneXus en los diferentes lenguajes: Java, C# y Ruby, según corresponda.
- Se realizan pruebas de integración del motor de reglas con el código generado GeneXus para los escenarios *Cliente*, *Producto* y *Factura* (ver Anexo E).
- Se resume el resultado de la evaluación.

4.1.2. Resultados de la Evaluación

Para la plataforma Java se evaluaron las herramientas *Jess* [44], *JenaRules* [41] y *Drools* [13].

Jena Rules [41], si bien es un framework específico para la Web Semántica, posee motores de inferencia que utiliza algoritmos de tipo forward y backward chaining. Se realiza una prueba de concepto con algunos casos de uso básicos, no calificando para los propósitos del proyecto debido a algunas restricciones encontradas, como por ejemplo, la inexistencia de mecanismos para declarar prioridades entre reglas. El siguiente texto detalla una de las restricciones que descalifica a la herramienta[1]; la no existencia de la propiedad *salience*.

> *The Jena rule engine was designed to make simple deductions, not transformations, and lacks the rich control features that fuller featured production systems have (such as rule salience).*

[1] `http://eprints.otago.ac.nz/164/1/dp2005-12.pdf`

Jess [44] es un motor de reglas que hace tiempo existe en el mercado; si bien no es open-source, es muy utilizado en el ambiente académico debido a que no tiene licencia en este caso. Para que una aplicación Java pueda interactuar con el intérprete *Jess* es necesario que los objetos que se insertan en la memoria de trabajo del motor satisfagan la especificación JavaBeans [38]. En particular, es necesario implementar la especificación *PropertyChangeListeners*[2]. Por defecto las clases generadas GeneXus que se asocian con los objetos sobre los cuales se pueden definir reglas no utilizan dicha especificación. Se agregó código manual para poder evaluar los escenarios planteados. Las pruebas realizadas fueron parcialmente satisfactorias, ya que no se logró implementar en su totalidad todos los escenarios.

Drools [13] es una herramienta open-source que hace varios años existe en el mercado; está respaldada por la empresa *RedHat* dentro de lo que es el área de desarrollo de middleware (*JBoss*). Se completaron en forma satisfactoria todos los escenarios de evaluación e investigaron además mecanismos de definición de reglas utilizando tablas de decisión y lenguajes específicos de dominio.

En la plataforma .Net según la pre-selección, se evaluaron *NXBre* [55] y *Drools.Net* [14] no teniendo en cuenta las propuestas comerciales.

Para el caso *NXBre* [55] no se pudo completar los escenarios de evaluación propuestos, pues no funcionaron los ejemplos de definición de reglas en lenguaje *RuleML* [86].

Para el caso *Drools.Net* [14], si bien es un port de *Drools 3.0*, se pudieron ejecutar los casos de uso propuestos.

Para *Ruby* sólo se hicieron pruebas con la gema *RuleBy* [85] ya que fue la única herramienta que calificó la pre-selección. Si bien no se implementaron todos los escenarios, el caso "*Hello World*" (ver Anexo G) funcionó correctamente. Se encontró una restricción cuando se definen reglas que condicionan y a la vez actualizan el mismo objeto; una misma regla se activa sistemáticamente causando que la ejecución quede en loop. Es posible solucionar el problema agregando nuevas propiedades a los objetos para utilizarse como bandera e indicar que una regla ya fue ejecutada (solución utilizada al evaluar *Jena Rules*).

Finalmente, la figura 4.1 detalla la selección de herramientas para las plataformas de ejecución GeneXus:

[2] `http://java.sun.com/j2se/1.5.0/docs/api/java/beans/package-summary.html`

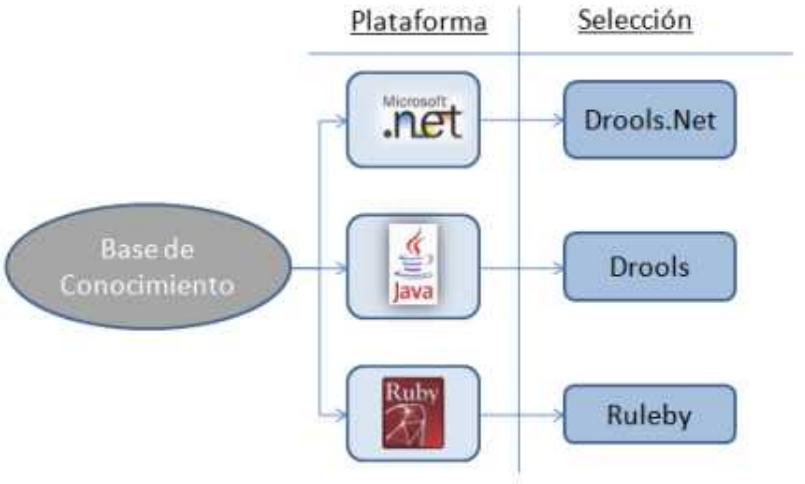


Figura 4.1.: Selección de productos por plataforma GeneXus

Para el caso Java, *Drools* [13] es el motor de reglas evaluado más completo y se completaron en forma satisfactoria todos los escenarios de evaluación.

Para el caso .Net se selecciona la herramienta *Drools.Net*.

Para el caso Ruby según la pre-selección de herramientas sólo calificó la gema *Ruleby*.

Se comienza utilizando como herramienta de prototipado el motor de evaluación de reglas Drools. 5.0 ya que es la herramienta open-source más completa. Por otra parte y de acuerdo con esta selección, se utiliza el generador GeneXus Java utilizando MySQL 5.0[3] como DBMS y Apache Tomcat 5.5[4] como contenedor de servlets.

4.2. Arquitectura

Se crean los siguientes objetos dentro del ecosistema GeneXus para representar el concepto de reglas:

- Objeto *Ruleset*: para encapsular la definición de reglas.
- Objeto *RuleDSL*: para capturar el lenguaje del experto del negocio.
- Objeto *RulesetEngine*: para encapsular y abstraerse de la funcionalidad provista por el motor de evaluación de reglas.

[3] `http://dev.mysql.com/downloads/mysql/5.0.html`
[4] `http://tomcat.apache.org/`

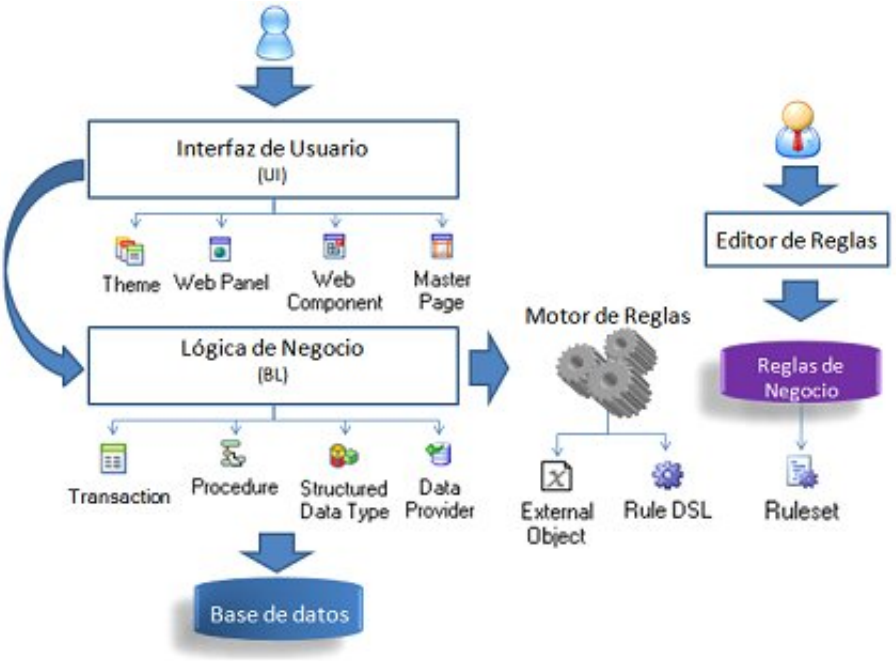


Figura 4.2.: Interacción de objetos GeneXus

A su vez se suscribe a diferentes eventos de disparo dentro de la infraestructura GeneXus para poder transformar los objetos GeneXus, en objetos dependientes de la plataforma que permitan interactuar con la infraestructura de definición de reglas existente. Al generarse y compilarse los objetos estándar GeneXus, se agregan funciones para traducir el lenguaje de reglas diseñado a la plataforma seleccionada, y, se generan además todas las dependencias necesarias para poder realizar modificaciones en ejecución.

4.3. Lenguaje Técnico de Reglas

Después de realizar un análisis de los diferentes lenguajes de reglas existentes, y vista la evaluación de motores de reglas realizada, se decide tomar un enfoque de construcción de un lenguaje de reglas en el ambiente GeneXus a nivel técnico, tal como se especifica por la mayoría de los motores de reglas existentes.

Se toma como referencia, por un lado el estándar PRR (Production Rule Repesentation) [71] y por otro lado, el lenguaje de definición de regla DRL (Drools Rule Language) [13], donde las reglas se definen en dos grandes bloques constructivos: "*Condiciones*" y "*Acciones*".

Supongamos la siguiente regla:

```
Clientes masculinos de hasta 25 años son de tipo A
```

que tiene la siguiente representación utilizando el lenguaje técnico de reglas en el contexto GeneXus (ver Anexo G):

```
rule "Clientes masculinos de hasta 25 años"
  when
    &c:Customer(CustomerGender = "Male", CustomerAge <= 25 )
  then
    &c.CustomerType = "A"
end
```

siendo `Customer` la representación GeneXus del cliente, `CustomerGender` y `CustomerAge` dos de sus atributos en donde se aplican las condiciones de la regla; `&c` la proyección del cliente en una variable que podrá utilizarse en la sección de acciones de la regla y `CustomerType` una propiedad del cliente que se modifica como consecuencia de la ejecución de la regla.

En este caso estamos utilizando el enfoque ECA Rules (Event Condition Action Rules) [118], donde:

- ***E**vento*: especifica cuándo se tienen que evaluar las reglas. En términos del prototipo se resuelve en forma procedural en el código GeneXus.
- ***C**ondición*: especifica el grupo de condiciones que deben de evaluarse para que la regla se pueda ejecutar, resuelto en la sección *when* de cualquier regla dinámica.
- ***A**cción*: especifica el grupo de acciones o consecuencias que se aplicarán cuando la regla se ejecute, resuelto en la sección *then* de cualquier regla dinámica.

4.3.1. Reglas

Dentro del contexto GeneXus, las reglas dinámicas se agrupan en el objeto *Ruleset*, definiendo los dos grandes bloques detallados anteriormente *When <Conditions>* y *then <Actions>*. La Figura 4.3 detalla el diálogo de creación de objetos del grupo *Rules* en donde se localiza el nuevo artefacto.

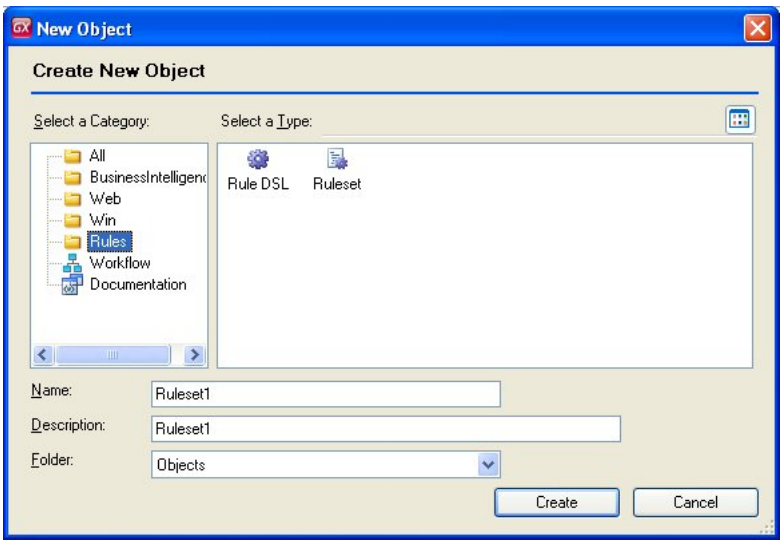


Figura 4.3.: Crear nuevo objeto Ruleset

Las reglas que se definan podrán actuar sobre cualquier objeto estructurado GeneXus, ya sea un *Business Component* como un *SDT*[5]. También se podrán ejecutar procedimientos que retornen una condición booleana como mecanismo de evaluación en la sección de condiciones.

4.3.2. Definición de una Regla

Se utiliza el esquema en el cual se divide la definición de una regla de negocio en dos bloques fundamentales:

```
when
  <condiciones>
then
  <acciones>
```

indicando que cuando ocurren ciertas *condiciones* se deben de aplicar ciertas *acciones*.

Una pregunta que surge al definir este esquema es ¿Por qué no se utilizó la opción *If* en vez de *when*?.

La cláusula *If* está fuertemente asociada al código procedural, mientras que la opción *when* tiene una semántica más declarativa, indicando que las condiciones no están "atadas" a una evaluación específica en el tiempo y puede ocurrir en cualquier momento durante la ejecución de la aplicación; es por eso que se selecciona *when* como palabra clave.

La sección *then* simplemente indica las acciones que deben ser ejecutadas cuando se cumplen las condiciones.

[5] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?6286`

Siendo un poco más estrictos, una regla tiene la siguiente estructura:

```
rule "Nombre De Regla"
  <Atributos De una Regla>
when
  <Sección izquierda o condiciones>
then
  <Sección derecha o acciones>
End
```

4.3.3. Partes de una Regla

Una regla especifica que cuando un conjunto de condiciones ocurre, especificado por lo que se denomina sección izquierda o condiciones (LHS - **l**eft **h**and **s**ide), entonces se debe de hacer lo que se especifica en la lista de acciones en la sección derecha de la regla (RHS - **r**ight **h**and **s**ide).

Una regla debe tener un nombre único que la identifica en la Base de Conocimiento.

La sección izquierda de la regla se especifica a continuación de la palabra clave `when`, mientras que la sección derecha se especifica a continuación de la palabra clave `then`.

Es importante destacar que no se pueden anidar reglas dado que cada regla debe ser atómica. La definición de la regla de negocio finaliza con la palabra clave `end`.

Sección Izquierda y Elementos Condicionales

La parte izquierda de las reglas es la sección donde se definen las condiciones. Consiste en cero o más elementos condicionales; en caso de no existir condiciones, implica que la regla siempre va a ser verdadera y por lo tanto se ejecutarán siempre las acciones. Supongamos la siguiente regla:

```
rule "Test"
  salience 10
  when
    // nothing
  then
    Msg( "Rule Engine is running ..." )
end
```

La regla de nombre `Test` mostrará un mensaje siempre que se evalúe el juego de reglas donde se encuentre dado que no tiene condiciones; notar que en la sección `when` se tiene un comentario (demarcado por //). Además, en este caso se está utilizando una propiedad denominada `salience` que declara en forma explícita la prioridad de la regla y se utiliza para ordenar las reglas cuando más de una es activada en el mismo evento de disparo (Ver Anexo C por más información).

Patrones El elemento patrón es el componente más importante dentro de lo que son los elementos condicionales para aplicar restricciones sobre reglas.

En su formato más simple, donde no se usan restricciones, un patrón "matchea" contra un hecho de un tipo determinado; en el contexto GeneXus será un SDT o BC.

En el siguiente ejemplo se tiene un Cliente (objeto `Customer`), por lo que el patrón "`&c:Customer()`" indica que se "matcheará" contra todos los clientes que existan en la memoria de trabajo y se proyecta el cliente en la variable `&c`.

```
rule "Customer Test"
  salience 10
  when
    &c : Customer( )
  then
    Msg( "just a Customer")
end
```

Dentro de los paréntesis es donde las condiciones se definen; es decir, es donde se deberán de agregar todas las restricciones que se quieran controlar sobre el objeto - en este caso el cliente.

Una restricción puede ser, un filtro sobre un campo, un grupo de filtros o condiciones sobre diferentes objetos. Las restricciones se separan entre ellas con los símbolos "`,`" (coma); "`and`" y "`or`".

Agreguemos más condiciones a la regla:

```
rule "Clientes masculinos de hasta 20 años son de tipo A."
  when
    &c : Customer( Customergender == "M", Customerage <= 20 )
  then
    Msg("Clientes masculinos de hasta 20 son de tipo A.")
    &c.Customertypeid = 1 // A
end
```

Se agregan dos restricciones sobre las propiedades `CustomerGender` y `CustomerAge` respectivamente.

Restricciones sobre Campos Especifica una restricción a ser utilizada sobre un campo. El campo que está siendo restringido puede asignarse a una variable en forma opcional si se desea proyectar su valor para tenerlo disponible tanto en la sección de acciones de la regla, como siguientes condiciones de la regla.

```
Customer( &cId:CustomerId == 5 )
Customer( &cTypeDsc:CustomerTypeDescription == "Type A")
```

Restricciones sobre literales Es la forma más simple de restricción y evalúa un campo contra el literal especificado, pudiéndose especificar un valor numérico, fecha, carácter o boolean.

Algunos ejemplos:

```
Customer( CustomerId == 5 )
Customer( CustomerDate < "27-Oct-2009")
Customer( CustomerTypeDescription == "Type A")
Customer( CustomerIsActive == true )
```

Si se proyectan los objetos en variables (*binding* de objetos en variables) para ser utilizados en la sección de acciones de las reglas, las restricciones se especifican de la siguiente forma:

```
&c:Customer( &cId:CustomerId == 5 )
&c:Customer( &cDate:CustomerDate < "27-Oct-2009")
&c:Customer( &cTypeDsc:CustomerTypeDescription == "Type A")
&c:Customer( &cIsActive:CustomerIsActive == true )
```

Las variables que se asocian a los hechos y sus campos se pueden utilizar en restricciones de campos siguientes:

```
&c : Customer( )
&i : Invoice( Customerid == &c.Customerid )
```

En este caso, en la primer línea se proyecta el cliente en la variable `&c`. En la segunda línea se asegura la condición que el cliente de la `Factura` sea el mismo que el `Cliente` proyectado en `&c` de la restricción anterior. Este es el mecanismo disponible para definir una regla que aplique restricciones sobre diferentes objetos del dominio.

Condicional *and* El elemento condicional `and` se utiliza para agrupar otros condicionales en una conjunción lógica.

```
Customer( CustomerId == 5 ) and Product( Productid == 4 )
```

En este caso se indica que la regla restringe al Cliente (`Customer`) con identificador 5 y el Producto (`Product`) con identificador 4. Esta regla es equivalente a especificarla como:

```
Customer( CustomerId == 5 )
Product( Productid == 4 )
```

donde el operador `and` se asume implícito.

Condicional *or* El elemento condicional `or` se utiliza para agrupar condicionales en disjunciones lógicas.

```
Customer( CustomerId == 5 ) or Product( Productid == 4 )
```

La regla restringe al cliente (`Customer`) con identificador 5 o el producto (`Product`) con identificador 4.

Condicional *eval* Permite ejecutar cualquier código en donde el resultado de la evaluación debe ser un boolean[6].

```
eval(&var1==&var2)
```

Condicional *not* Se corresponde con el cuantificador no existencial de la lógica de primer orden, chequeando la no existencia de un elemento en la memoria de trabajo.

```
not Customer()
```

Recordar que los motores de ejecución asumen un universo cerrado (CWA - Closed Word Assumption) [155], por lo que la regla tendrá la posibilidad de ser ejecutada cuando no exista ningún cliente en la memoria de trabajo.

Otro ejemplo agregando restricciones sobre el producto:

```
not ( &p : Product( Productid == 6 ))
```

especifica que se activará la regla para cualquier producto (`Product`) que no esté identificado con el valor 6.

Condicional *exists* Se corresponde con el cuantificador existencial de la lógica de primer orden, chequeando la existencia de un elemento en la memoria de trabajo.

Cuando se utiliza este patrón, la regla será disparada solamente una vez, sin importar la cantidad de elementos que existan en la memoria de trabajo que coincidan con la regla.

Dado que solo importa la existencia del hecho, no se permiten realizar bindings dado que no tiene sentido la proyección de ningún objeto en este caso.

```
exists( Customer())
```

La regla tendrá la posibilidad de ser ejecutada cuando exista al menos un cliente.

Otro ejemplo con varias restricciones sobre un campo del producto:

```
exists ( &p : Product( Productid >= 4 , Productid <= 5 ))
```

especifica que se activará la regla si existe algún producto (`Product`) con un identificador comprendido en el rango 4-5.

[6] no se implementa en su totalidad, consultar el Anexo F por más información.

Sección Derecha

La sección derecha de una regla es básicamente un bloque de código procedural a ser ejecutado. Se recomienda no utilizar código imperativo o condicional (una regla debería ser atómica por naturaleza), aunque esto no es una restricción.

En el contexto de ejecución GeneXus se soporta definir lo siguiente:

- Manipulación de elementos de SDTs (objetos estructurados) y BCs (Business Components) previa proyección de los mismos en la sección izquierda de una regla o mediante la creación en forma explícita de un objeto mediante el uso del operador `new`. Algunos ejemplos son:

```
// Asignación básica
&c.CustomerId = 2
&c.Customertypeid = 2 // B

// Uso del operador new para crear un objeto
&itemInvoiceLine8 = new Invoice.Line()

// Algunas asignaciones
&itemInvoiceLine8.InvoiceLineId = 100
&itemInvoiceLine8.ProductId = 6
&itemInvoiceLine8.InvoiceLineQty = 1
&itemInvoiceLine8.InvoiceLineProductPrice= 1.00
```

- Asignación de cualquier tipo de expresión aritmética que utilice los operadores +,-,*, %, por ejemplo:

```
&p.Productprice = &p.Productprice + &p.Productbase * 0.01
```

- Operadores específicos para interactuar con el motor de reglas, como por ejemplo `update`, `insert`, `retract`; ver sección 4.5.

```
// inserción de un objeto en la memoria de trabajo
insert( &itemInvoiceLine8 )

// interación con el motor de evaluación
RuleEngine.Halt()
```

- Función `Msg` para mostrar por salida estándar un mensaje.

```
Msg("Hola Mundo") // Uso de Función Msg
```

- Sentencia de control `If`.

```
// Uso de sentencia del control If
if (1==1)
  &c.CustomerId = 2
else
  &c.CustomerId = 1
endif
```

- Ejecución de procedimientos GeneXus.

```
// Llamada a un objeto GeneXus
gxProcedure01.Call(&i8,&itemInvoiceLine8,&cId,&iAmount,&iDate,&cName,true)
gxLog.Call(&i8.InvoiceId,"Invoice02 - Call procedure")
```

Operadores específicos - sección izquierda de Reglas

Además del código procedural detallado anteriormente que puede ser ejecutado en el contexto de acciones de una regla, se disponen de operadores específicos para poder interactuar con el motor de evaluación de reglas.

- *RuleEngine*: provee un acceso directo al motor de evaluación de reglas.

```
//procesar grupo de reglas "descuentos"
RuleEngine.Focus( "descuentos" )

// agregar un error
RuleEngine.AddError(1,"Menores de 21 no pueden realizar compras")

// interación con el motor de evaluación
RuleEngine.Halt()
```

- *InsertLogical*: define una inserción lógica de un hecho.

```
&vAlert = new Alert()
insertLogical( &vAlert )
```

RuleEngine La palabra clave `RuleEngine` es un acceso directo al motor de reglas que se dispone desde el contexto de ejecución de consecuencias de reglas.

Se implementan las siguientes operaciones:

- *Halt*: permite detener la ejecución del motor de reglas. El motor de reglas limpia la agenda que tiene lista de reglas que se deberían haber ejecutado devolviendo el control a la aplicación; generalmente se utiliza para realizar validaciones.
- *AddError*: representa la inserción de un error dentro del contexto de evaluación de reglas.
- *Focus*: restringe el conjunto de reglas a evaluarse a un grupo específico que utilicen la propiedad "`Agenda Group`" (ver Anexo C por más información). Notar que la forma de procesar los grupos de reglas por el motor de evaluación es un stack, la última declaración es la primera en ser evaluada. Supongamos que desde una regla se especifica lo siguiente:

```
RuleEngine.Focus( "discounts" )
RuleEngine.Focus( "promotions" )
RuleEngine.Focus( "validate" )
```

El motor de evaluación de reglas va a aplicar las reglas teniendo en cuenta primero el grupo de validaciones (`validate`), posteriormente se evalúa el grupo de promociones (`promotions`) y finalmente el grupo de descuentos (`discounts`). Se detalla un caso de uso utilizando esta propiedad en el Anexo G.

Un ejemplo de uso puede ser por ejemplo si se tiene el siguiente bloque de código en las consecuencias de una regla de validación:

```
Msg("Un Cliente menor a 21 años no puede realizar compras")
RuleEngine.AddError(1,"El Cliente no puede realizar compras")
RuleEngine.Halt()
```

Inicialmente se envía un mensaje detallando la situación, posteriormente se utiliza la operación `AddError` para indicar que se está violando una validación del modelo de negocio, y finalmente, utilizando la operación `Halt`, se finaliza en forma explícita la evaluación de reglas y se devuelve el control al objeto llamador.

InsertLogical Permite especificar dependencias lógicas entre hechos. Todos los hechos que se insertan en la sección derecha de una regla se torna dependiente de la sección izquierda (sección de "matcheo"). En caso que alguno de los patrones de "matcheo" se tornen inválidos, los hechos dependientes se eliminan en forma automática.

Propiedades que debe cumplir un hecho lógico:

- El objeto es único.
- La validez del objeto es mantenida en forma automática por el motor de evaluación.

Se basa en la premisa de que un objeto sólo puede existir mientras una sentencia sea verdadera especificada a través de una regla. Desde la sección de consecuencias de una regla se puede insertar en forma lógica un objeto (utilizando el operador *InsertLogical*), lo que significa que el objeto que se inserta solo va a permanecer en la memoria de trabajo mientras que la regla que hizo la inserción lógica sea verdadera. Cuando dicha regla no sea verdadera, el motor de reglas automáticamente elimina de la memoria de trabajo el objeto insertado en forma lógica (operador *InsertLogical*), causando un nuevo ciclo de evaluación de reglas.

Un objeto lógico puede tener más de un justificativo. En caso que se deduzca desde varias reglas, el mismo será mantenido en la memoria de trabajo mientras al menos una condición de todos sus justificativos sea verdadera.

La operación solo puede ejecutarse desde la consecuencia de una regla de la siguiente forma:

```
insertLogical(&c)
```

siendo `&c` un objeto estructurado GeneXus de tipo `Cliente`.

Ejemplo:

```
rule "Mensaje Incorrecto"
  salience 10
  when
    exists( &m:MyMessage( Status == false ) )
  then
    &vAlert = new Alert()
    insertLogical( &vAlert )
end
```

Por un caso de uso utilizando ésta operación consultar el caso de uso *Truth Maintenance* [142] en el Anexo G.

4.4. Lenguaje de Alto nivel de Reglas

El lenguaje técnico de reglas diseñado, si bien tiene una gran expresividad, su uso queda circunscripto al desarrollador GeneXus dado que es necesario un conocimiento de la sintaxis y semántica específica que se maneja en la Base de Conocimiento GeneXus.

Para poder brindar un lenguaje de alto nivel de abstracción de reglas de negocio que sea fácilmente entendible por el experto del negocio, es deseable que su especificación se acerque lo más posible al *lenguaje natural*.

Para ello se toman ideas de lo que es el enfoque DSL según detalla la sección 3.5.5. Un DSL (Domain Specific Language) [169] es un lenguaje de programación de expresividad limitada que se centra en un problema de dominio particular.

Este DSL específico de reglas extiende lenguaje de reglas "técnico" para que sea fácilmente entendible por el experto del dominio, ocultando así las particularidades de lo que es el lenguaje técnico.

Es una capa de abstracción del lenguaje de reglas, donde se permite utilizar toda la infraestructura de reglas creada.

El implantador puede definir el lenguaje de modelado que contenga los conceptos y reglas que sean necesarios para los expertos del negocio, especificando internamente la transformación (mapping) al lenguaje de reglas GeneXus diseñado (detallado en la sección 3.5.5).

4.4.1. RuleDSL

Se busca que el lenguaje de reglas se pueda parametrizar lo máximo posible, intentando acercarlo al lenguaje natural.

Se debe analizar cómo describe la regla el experto en el negocio, qué palabras utiliza, y posteriormente relacionar estos conceptos al modelo de objetos (Mapping entre BOM y XOM detallado en la sección 3.5.5), utilizando los constructores de reglas existentes.

Seguramente, el lenguaje específico de dominio para reglas evolucionará a medida que aumenten las reglas, y su funcionamiento también mejorará en la medida que se utilicen términos comunes en forma repetitiva y se parametricen en el lenguaje.

Se crea un nuevo objeto GeneXus denominado *RuleDSL* para administrar lenguajes específicos de dominios en el contexto de reglas de negocio y se agrega un mecanismo para indicar que un juego de reglas (representado por el objeto *Ruleset*) utiliza dichos lenguajes.

4.4.2. Objeto RuleDSL

El objeto GeneXus *RuleDSL* puede servir como una capa de abstracción que permita separar la creación de reglas, de los objetos del dominio sobre los que se actúa al evaluar las reglas; también pueden actuar como plantillas predefinidas de condiciones y acciones a definirse sobre las reglas.

Si las reglas tienen que ser leídas y validadas por personal no técnico, el recurso *RuleDSL* permitirá esconder los detalles de implementación, enfocándose en lo que es la "regla de negocio" en sí. También permitirá editar en forma muy controlada las reglas existentes.

Es importante destacar que este recurso no tiene impacto sobre lo que es la ejecución de reglas en runtime. Dada una especificación de regla en formato DSL, internamente al salvar las reglas se utiliza el algoritmo de Markov [52] para realizar la traducción del lenguaje de alto nivel al técnico. Una vez que se obtiene la representación en formato técnico GeneXus, se utiliza el mecanismo de parsing desarrollado para generar la representación de la regla en la plataforma destino.

4.4.3. Editor RuleDSL

El objeto *RuleDSL* se accede a través del dialogo de creación de objetos (ver Figura 4.3) y consta de 3 columnas para definir las expresiones válidas del lenguaje:

- *Language Expression*: lenguaje natural para el experto del negocio.
- *Mapping Expression*: lenguaje técnico.
- *Scope*: alcance de la expresión.

Para el caso de uso relacionado con el *Cliente* (ver sección 5.2.2) se creó el siguiente objeto *RuleDSL*:

Language Expression	Mapping Expression	Scope
Customer gender is "{gender}" and age is less than '{age}'	&c : Customer(CustomerGender == "{gender}" , CustomerAge <= {age})	CONDITION
Customer age is greater than '{age}'	&c : Customer(CustomerAge > {age})	CONDITION
Customer gender between '{startAge}'and '{finishAge}'	&c : Customer(CustomerAge >= {startAge} , CustomerAge <= {finishAge})	CONDITION
Customer Type is '{type}'	&c : Customer(CustomerTypeid == {type})	CONDITION
- From Montevideo	CountryId == 1,CityId == 1	CONDITION
There is a Customer	&c : Customer( )	CONDITION
Log : "{message}"	Msg("{message}")	CONSEQUENCE
Set Customer Type to '{value}'	&c.CustomerTypeid = {value}	CONSEQUENCE
Set Customer Maximum Allowed to '{value}'	&c.CustomerMaxAllowed = {value}	CONSEQUENCE
Update the fact : '{variable}'	update(&{variable})	CONSEQUENCE

El editor DSL provee un formato tabular para mapear el lenguaje a expresiones de reglas.

La sección "*Language Expression*" detalla exactamente qué expresiones se manipulan en las reglas. Este objeto se utiliza también como entrada al asistente de contenido permitiendo utilizar la opción *Ctrl+Space* para seleccionar las expresiones del lenguaje disponibles para manipular las reglas, como detalla la Figura 4.4.

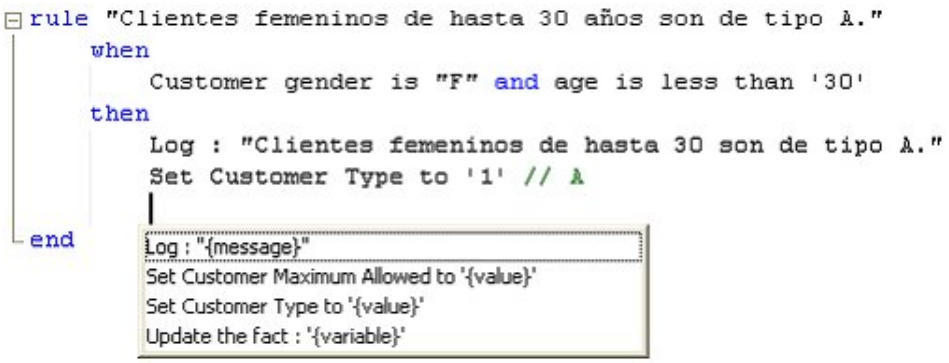


Figura 4.4.: Asistente de contenido para reglas "alto nivel"

La columna "*Mapping Expression*" define el lenguaje técnico de reglas que realmente se utilizará cuando se use la regla y referencia objetos GeneXus, propiedades y relaciones que se definan en la Base de Conocimiento.

Notar que estas expresiones son diferentes dependiendo si las mismas se utilizan en las condiciones o consecuencias de la regla.

Es para ello que la tercera columna "*Scope*" indica en qué sección la expresión es válida. Se dispone de las siguientes opciones:

- *CONDITION*: la expresión es válida en la sección de condiciones.
- *CONSEQUENCE*: la expresión es válida en la sección de acciones.
- *KEYWORD*: la expresión es válida para cualquier palabra clave.
- *ANY*: la expresión es válida en cualquier sección.

4.4.4. Funcionamiento

Como se detalló anteriormente, la traducción de reglas desde el formato "alto nivel" al técnico se realiza utilizando ideas del algoritmo de Markov [52].

El algoritmo de Markov utiliza un conjunto de reglas de sustitución a aplicarse en forma sistemática sobre un string de entrada. Las reglas de sustitución son una lista de pares de string que se representan de la forma:

```
patrón → reemplazo
```

En el objeto GeneXus *RuleDSL*, `patrón` se corresponde con la columna *Language Expression* y `reemplazo` se corresponde con *Mapping Expression*.

A partir de un string de entrada que será un juego de reglas en formato "*alto nivel*", se realizan las siguientes operaciones:

1. Se chequean las reglas de traducción del objeto *RuleDSL* en el orden que fueron definidas de arriba hacia abajo para verificar si alguno de los patrones se encuentran en el string de entrada.
2. En caso negativo, el algoritmo finaliza.
3. Si se encuentra uno o más patrones, se reemplaza el texto izquierdo que "matchea" con su reemplazo.
4. Si la regla aplicada es la última, se termina el algoritmo.
5. Se vuelve al paso 1 del algoritmo.

El experto de negocio manipula las reglas en términos de lo que indique el objeto *RuleDSL* en particular en la columna "*Language Expression*" y salva la definición de reglas.

En términos generales, se llega al lenguaje técnico a partir del lenguaje de alto nivel mediante las reglas de sustitución que detalla el objeto *RuleDSL* correspondiente. Sistemáticamente, cada sustitución transforma una parte del lenguaje de "alto nivel"

al "técnico", y al finalizar el algoritmo se obtiene la representación en formato técnico para el juego de reglas parámetro. La Figura 4.5 ejemplifica este proceso:

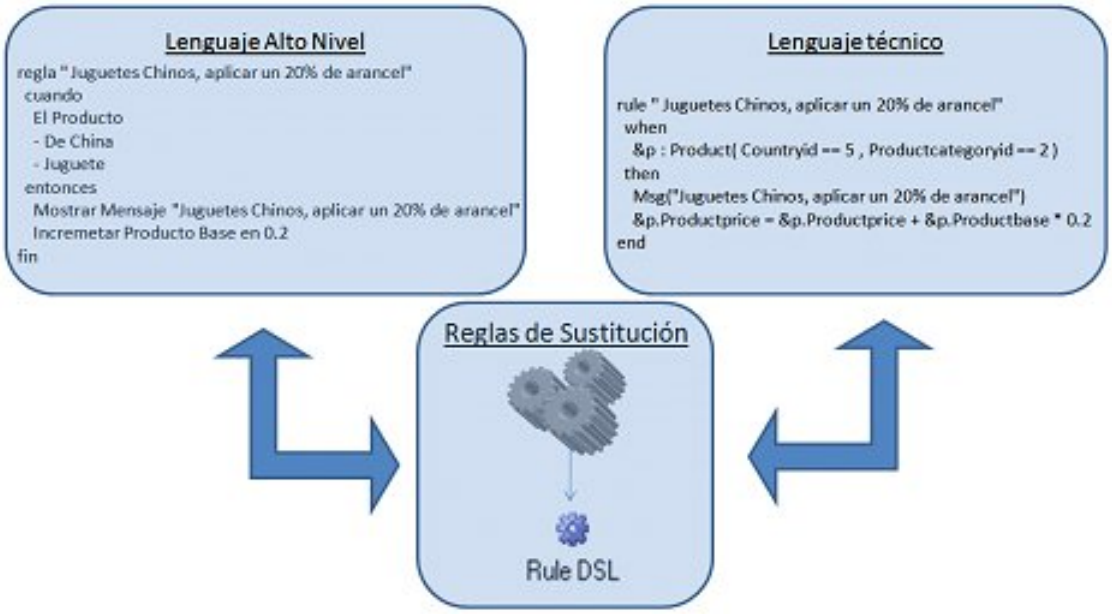


Figura 4.5.: Reglas de sustitución para traducir lenguajes de reglas

El mecanismo de traducción se activa cuando las reglas se salvan. Se utiliza la configuración de un *RuleDSL* siempre y cuando en el cabezal del objeto de reglas exista la palabra clave: *expander*, es decir, defina lo siguiente:

```
expander <RuleDSLName>
```

Se busca el objeto *RuleDSL* de nombre "*RuleDSLName*" para que el algoritmo de traducción pueda expandir las expresiones en el lenguaje de dominio definidas y logre la representación equivalente para la plataforma GeneXus.

El traductor se encarga de leer la definición del lenguaje correspondiente (objeto *RuleDSL*), línea a línea, y trata de "matchear" las "*Language Expression*" – según el alcance – al lenguaje técnico que se detalla en la columna "*Mapping Expression*". El proceso implica utilizar expresiones regulares en dos etapas: para realizar el "matcheo" de patrones (una expresión se utiliza en un juego de reglas) y aplicar la sustitución correspondiente (obtener la representación equivalente en la plataforma GeneXus).

Notar que tanto las expresiones del lenguaje como la traducción (columnas *Language* y *Mapping Expression*) utilizan las llaves ("*{}*") para demarcar el uso de variables. Estos textos se corresponden con valores instanciados en cada regla y serán mantenidos en el proceso de traducción.

Para fijar ideas y entender como funciona el algoritmo, se detalla paso a paso las sustituciones que se realizan para lograr la representación técnica de la regla de la

Figura 4.4 utilizando las reglas de sustitución de la sección 4.4.3. La regla define una única condición que es:

```
Customer gender is "F" and age is less than '30'
```

Al aplicar el algoritmo de traducción, el proceso de *pattern matching* de expresiones detecta que se tiene que aplicar la siguiente sustitución en el contexto de las condiciones de la regla (columna *Scope*):

```
Customer gender is "{gender}" and age is less than '{age}'
→
&c : Customer(CustomerGender == "{gender}", CustomerAge <= {age})
```

Al realizar la sustitución de parámetros se obtiene el lenguaje técnico:

```
&c : Customer( CustomerGender == "F", CustomerAge <= 30 )
```

Notar que la sustitución transformó la condición a dos restricciones sobre el objeto `Customer`; las variables `{gender}` y `{age}` mantienen los valores "`F`" y '`30`' respectivamente.

El algoritmo continúa ejecutando, se procesan las acciones de la regla; la primera acción es:

```
Log : "Clientes femeninos de hasta 30 son de tipo A."
```

Se sustituye utilizando el patrón:

```
Log : "{message}"
→
Msg("{message}")
```

Notar que en éste patrón se define una única variable `{message}` con el valor "`Clientes femeninos de hasta 30 son de tipo A.`", obteniéndose el lenguaje técnico:

```
Msg("Clientes femeninos de hasta 30 son de tipo A.")
```

Para finalizar el procesamiento de la regla, se procesa la acción:

```
Set Customer Type to '1'// A
```

Se sustituye utilizando el siguiente patrón y mantiene el valor que define la variable *{value}*:

```
Set Customer Type to '{value}'
→
&c.CustomerTypeid = {value}
```

obteniéndose el lenguaje técnico:

```
&c.CustomerTypeid = 1 // A
```

El resultado final después de aplicar el algoritmo de Markov [52] y según la transformación realizada en tres iteraciones es la regla de la Figura 4.6.

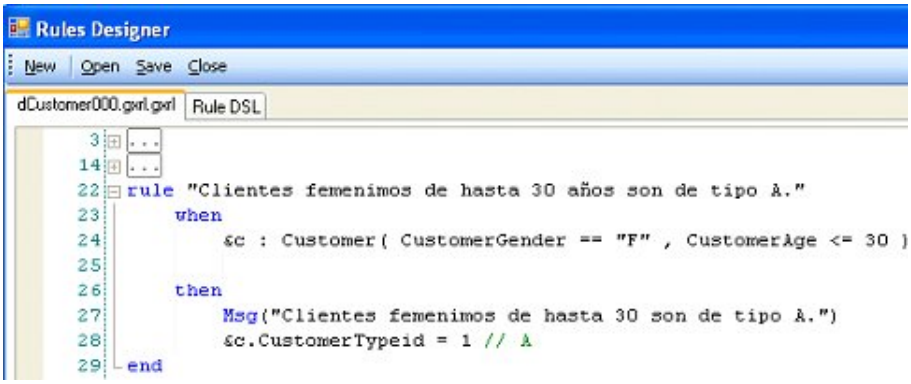


Figura 4.6.: Resultado de la traducción de una regla de negocio a formato técnico

Es importante resaltar que los espacios se deben tener en cuenta cuando se está procesando el lenguaje *RuleDSL*, donde las líneas se procesan en forma individual. Recordar que el mecanismo utilizado para traducir de lenguaje de dominio a técnico es una macro-sustitución utilizando expresiones regulares; es conveniente utilizar la facilidad de *intellisense*[7] para definir las reglas de negocio y asegurar que se usan correctamente los bloques constructivos del lenguaje.

Además esto brinda otra ventaja. El experto del negocio, al utilizar este recurso tanto a nivel de condiciones como consecuencias, obtiene fácilmente el conjunto de operaciones y restricciones que puede agregar para influir en el comportamiento de la aplicación que está ejecutando.

No es necesario definir todas las expresiones válidas del lenguaje en el objeto *RuleDSL* correspondiente, una regla de negocio podrá componerse de expresiones del lenguaje de alto nivel y técnico, por lo que si se quiere saltear la sustitución de parámetros en una línea particular, utilizar el prefijo "`>`" al inicio de cada línea. Con este recurso se pueden lograr representaciones híbridas de reglas.

Siguiendo el ejemplo, una representación de regla en formato híbrido equivalente a la regla de la Figura 4.4 es:

[7] `http://en.wikipedia.org/wiki/IntelliSense`

```
rule "Clientes femeninos de hasta 30 son de tipo A."
  when
    Customer gender is "F" and age is less than '30'
  then
    > Msg("Clientes femeninos de hasta 30 son de tipo A.")
    > &c.CustomerTypeid = 1 // A
end
```

El proceso de traducción obtiene la misma representación técnica de regla que detalla la Figura 4.6. En este caso se mantiene el lenguaje de alto nivel en las condiciones, sin embargo la sección de acciones utiliza directamente el lenguaje técnico.

4.4.5. Agregando restricciones a Hechos

Al definir condiciones sobre reglas, seguramente sea importante agregar varias restricciones en la sección izquierda (LHS) sobre un mismo hecho.

Un hecho puede tener decenas de propiedades; el problema surge en cómo definir restricciones sobre un hecho utilizando *RuleDSL*.

Se utiliza la siguiente convención: cualquier expresión *RuleDSL* que comience con un "–" (guión) se asume que será una restricción, que deberá ser agregada al primer hecho en las líneas superiores de la condición donde aparezca.

Supongamos que se tiene la siguiente regla (ver el caso de uso relacionado con productos en la sección 5.2.3):

```
regla "Juguetes Uruguayos, no aplicar arancel"
  cuando
    El Producto
     - Juguete
     - De Uruguay
  hacer
    Mostrar Mensaje "Juguetes Uruguayos, no aplicar arancel"
    Asignar Precio Base
fin
```

y el siguiente *RuleDSL*:

Language Expression	Mapping Expression	Scope
El Producto	&p : Product( )	CONDITION
- Juguete	Productcategoryid == 2	CONDITION
- De Uruguay	Countryid == 1	CONDITION
Mostrar Mensaje "{m}"	Msg("{m}")	CONSEQUENCE
Asignar Precio Base	&p.Productprice = &p.Productbase	CONSEQUENCE
fin	end	KEYWORD
regla	rule	KEYWORD
hacer	then	KEYWORD
cuando	when	KEYWORD

El algoritmo de sustitución va a reconocer las líneas que comienzan con "–" (es obligatorio que se especifiquen en diferentes líneas) y se agregan las restricciones a la declaración superior.

En este caso las restricciones "`Juguete`" y "`De Uruguay`" comenzando con el carácter "`-`" se deberán aplicar sobre "`El Producto`". El resultado de la traducción es el siguiente:

```
rule "Juguetes Uruguayos, no aplicar arancel"
  when
    &p : Product( Countryid == 1 , Productcategoryid == 2 )
  then
    Msg("Juguetes Uruguayos, no aplicar arancel")
    &p.Productprice = &p.Productbase
end
```

Notar en este caso que también se procesaron las palabras clave `regla`, `cuando`, `hacer` y `fin` que se utilizan como alias para lograr una definición de reglas similar al lenguaje español. Es decir, partiendo de la estructura fundamental de la regla con la sustitución simple de palabras claves (columna *Scope* del objeto *RuleDSL*):

```
regla     →    rule
hacer     →    then
cuando    →    when
fin       →    end
```

se obtiene un lenguaje totalmente parametrizado como se detalla en la Figura 4.7.

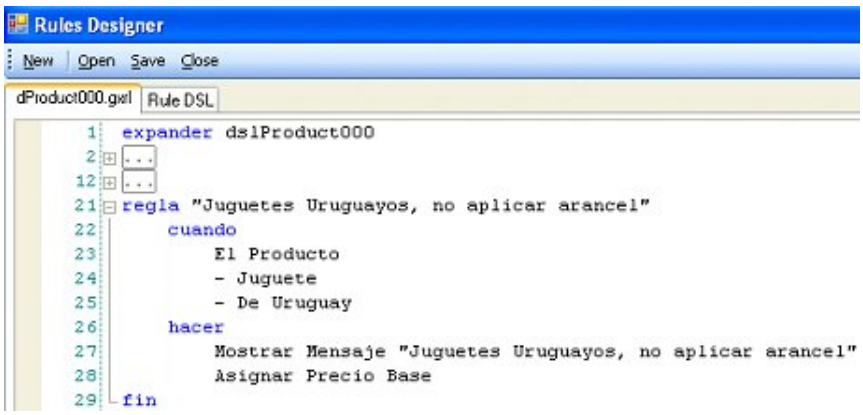


Figura 4.7.: Regla de negocio “alto nivel” en español

4.5. Motor de Evaluación

La representación del motor de evaluación de reglas se realiza utilizando un objeto externo GeneXus denominado *RulesetEngine*.

Este objeto externo (*External Object*) es el envoltorio (wrapper) del motor de ejecución de reglas utilizado para la plataforma Java (Drools 5.0) en el contexto de ejecución GeneXus. El objeto expone las principales funcionalidades para tener capacidades de evaluación de reglas en tiempo de ejecución.

La interfaz del objeto a nivel de abstracción GeneXus se detalla en la Figura 4.8.

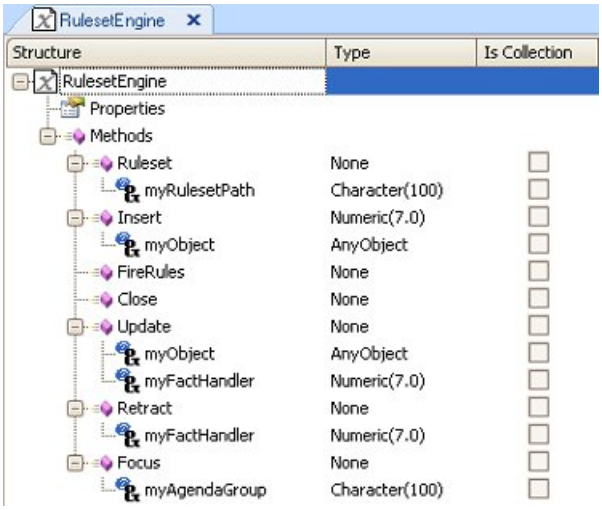


Figura 4.8.: Objeto GeneXus RulesetEngine

4.5.1. Métodos

El objeto *RulesetEngine* disponibiliza los siguientes métodos detallados en la Figura 4.9.

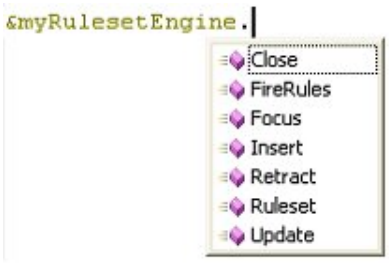


Figura 4.9.: Métodos del objeto RulesetEngine

- Ruleset.
- Insert.
- Update.
- Retract.
- Focus.
- FireRules.
- Close.

Ruleset

Informa al motor de reglas el nombre del juego de reglas que se va a utilizar. Se corresponde con un objeto *Ruleset* existente en la KB o creado en tiempo de implantación o parametrización con el diseñador de reglas en runtime (detallado en la sección 4.8).

Ejemplo:

```
&re.Ruleset("CustomerDynamicRules")
```

siendo la variable `re` de tipo `RulesetEngine` (representando el motor de reglas) y `CustomerDynamicRules` el nombre de un objeto *Ruleset*.

Insert

La operación `Insert` (también comúnmente denominado `assert`) representa el proceso de agregar objetos a la memoria de trabajo del motor de reglas.

Es el mecanismo que se provee sobre el cual se informa al motor de ejecución de reglas de la existencia de un nuevo hecho. Cuando esto ocurre, se examina en la memoria de trabajo el juego de reglas existente. Aquellas que tienen que ejecutarse se marcan, pero no serán ejecutadas hasta que explícitamente se ejecute el método `fireAllRules`.

Todo el trabajo de decidir cuáles reglas son las que se deben disparar se realiza en tiempo de inserción de un hecho.

Ejemplo de uso:

```
&ret = &re.Insert(&cliente) // insert object to the Engine
```

siendo la variable `re` de tipo `RulesetEngine` (representando el motor de reglas) y `&cliente` un objeto GeneXus de tipo `Cliente`.

Es importante resaltar que cuando se inserta un objeto se retorna un entero que representa el identificador del objeto en la memoria de trabajo del motor comúnmente denominado *Fact Handler*.

Dicho identificador es utilizado para referenciarlo posteriormente desde código procedural, por ejemplo si se quisiera eliminar o modificar utilizando las operaciones `retract` y `update` respectivamente.
Dentro del contexto de ejecución de consecuencias de una regla, la operación `insert` se puede ejecutar de la siguiente forma:

```
insert(&c)
```

siendo `&c` un objeto GeneXus de tipo `Cliente`.

Retract

La operación `Retract` elimina objetos de la memoria de trabajo, generando las siguientes consecuencias en el contexto de ejecución y activación de reglas:

- Aquellas reglas que fueron activadas y están prontas para ser ejecutadas se eliminan de la agenda.
- Cualquier acción en la agenda que involucre al objeto se elimina en forma automática de la misma.
- Cualquier regla que se encuentra activa y depende del hecho que se está eliminando se cancela.

Notar que es posible definir reglas que dependen de la no existencia de dicho hecho (utilizando los operadores `not` y `exist`), en este caso al eliminar el hecho, puede causar que este tipo de regla se active.

La operación puede ser ejecutada tanto desde el contexto de ejecución procedural GeneXus, como dentro de la sección de consecuencias de una regla.

En el contexto de ejecución de consecuencias de una regla, se puede utilizar de la siguiente forma:

```
retract(&c)
```

siendo `&c` un objeto GeneXus de tipo `Cliente` proyectado en la sección de condiciones de la regla.

Para utilizar la operación desde el código procedural, es necesario conocer el *FactHandler* que identifica al objeto en la memoria de trabajo. El *FactHandler* de un objeto se obtiene cuando se inserta el objeto en la memoria de trabajo.

Supongamos el siguiente código GeneXus:

```
&ret = &re.Insert(&cliente)
...
&re.Retract(&ret)
```

siendo la variable `re` de tipo `RulesetEngine` (representando el motor de reglas y una sesión de trabajo activa) y `&ret` el identificador de `&cliente` en la memoria de trabajo. La primer línea representa la inserción del cliente al mecanismo de evaluación siendo `&ret` el *FactHandler* asociado; en la última línea se elimina el objeto de la memoria de trabajo utilizando la variable `&ret`.

Update

La operación `Update` permite actualizar un objeto ya insertado en la memoria de trabajo; es el mecanismo existente por el cual se le notifica al motor de reglas que un objeto cambió, por lo que serán reevaluadas las reglas que tengan en cuenta dicho objeto.

La operación puede ser ejecutada tanto desde el contexto de ejecución GeneXus como dentro de la sección de consecuencias de una regla.

En el contexto de ejecución de consecuencias de una regla, se puede utilizar de la siguiente forma:

```
&c.ClienteTipo = 3
...
update(&c)
```

siendo `&c` un objeto GeneXus `Cliente` proyectado desde la sección de condiciones de la regla. Se modifica la propiedad `ClienteTipo` del mismo y posteriormente se informa al motor de reglas de la actualización del objeto de manera de reevaluar reglas que involucren al objeto `&c`.

Para utilizar la operación desde el código procedural, al igual que con la operación `Retract` (detallada en la sección anterior) es necesario conocer el *FactHandler* asociado al objeto cuando el mismo fue insertado en el contexto de ejecución de reglas, supongamos el siguiente código GeneXus:

```
&ret = &re.Insert(&cliente)
...
&re.Update(&cliente,&ret)
```

siendo la variable `re` de tipo `RulesetEngine` (representando el motor de reglas y una sesión de trabajo activa), `&cliente` un objeto GeneXus de tipo `Cliente` y `&ret` el identificador de `&cliente` en la memoria de trabajo (*FactHandler*) obtenido como resultado de la operación `InsertObject`; en la última línea se actualiza el objeto `Cliente` utilizando la variable `&ret`.

Focus

Configura el foco en un grupo específico de reglas, detallado por la propiedad `Agenda-group` para cada regla atómica.

```
&re.Focus("discouts")
```

siendo la variable `re` de tipo `RulesetEngine` (representando el motor de reglas y una sesión de trabajo activa), `discounts` un grupo de activación válido. En el Anexo G sección G.2 se detalla un ejemplo de facturación en donde las reglas se agrupan en tres grupos: validaciones, promociones y descuentos.

FireRules

Se ejecutan las reglas de negocio definidas en el objeto *Ruleset* para los objetos que fueron insertados en la memoria de trabajo del motor.

```
&re.FireRules()
```

siendo la variable `re` de tipo *RulesetEngine* (representando el motor de reglas y una sesión de trabajo activa) y `FireRules` el método para indicar que se disparen las reglas.

Close

Se cierra la sesión con el motor de reglas y continúa la ejecución procedural.

```
&re.Close()
```

siendo la variable `re` de tipo *RulesetEngine* y `Close` el método para cerrar la sesión.

4.6. Funcionamiento General

Hasta el momento se han definido todos los artefactos involucrados en la evaluación de reglas.

Existe un lenguaje de alto nivel de abstracción (representado por el objeto GeneXus *RuleDSL*) para las reglas de negocio que mapea a un lenguaje técnico de reglas dinámicas GeneXus (objeto GeneXus *Ruleset*) y permite especificar reglas sobre objetos estructurados GeneXus.

En tiempo de diseño en el IDE GeneXus, se deberán crear objetos *Ruleset* donde se encapsulan las reglas dinámicas utilizando el enfoque PRR [71] de definición de reglas en los cuales por cada regla se definen dos grandes bloques: *condiciones* que determinan cuando aplica una regla y *acciones* que definen las operaciones a ejecutar si la regla se tiene que disparar.

Mediante el objeto externo *RulesetEngine* se permite utilizar todas las operaciones del motor de evaluación de reglas desde la lógica de negocio GeneXus.

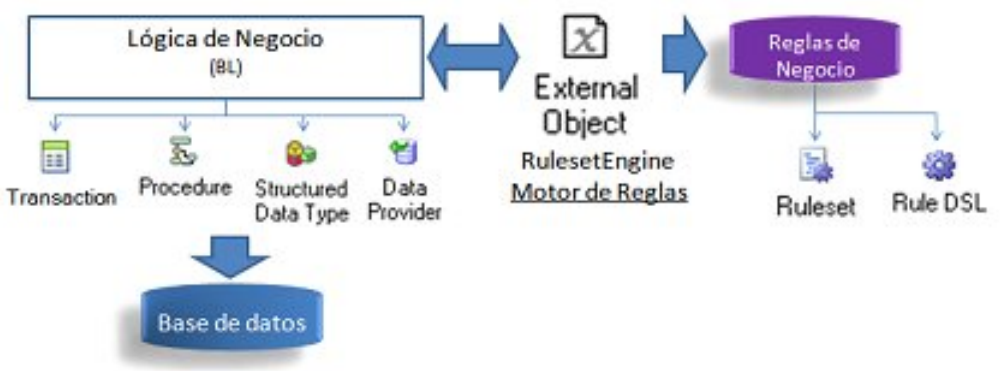


Figura 4.10.: RulesetEngine y demás objetos

Para poder interactuar con el motor de reglas y lograr evaluar un juego de reglas encapsulado en un objeto *Ruleset*, es necesario agregar al menos el siguiente código procedural en cualquier objeto GeneXus:

```
&myRulesetEngine.Ruleset(&ruleset)
&ret = &myRulesetEngine.Insert(&myCustomer)
...
&myRulesetEngine.FireRules()
...
&myRulesetEngine.Close()
```

Siendo `myRulesetEngine` una variable de tipo *RulesetEngine*; `&ruleset` la variable que contiene el nombre del juego de reglas a evaluar (objeto *Ruleset* en la KB GeneXus) y `&myCustomer` el cliente correspondiente que se inserta en la memoria

de trabajo del motor de evaluación. Si es necesario trabajar con más objetos, la operación `Insert` se deberá de repetir por cada objeto.

Para lograr el dinamismo deseado, es decir modificar reglas de negocio en tiempo de ejecución, es necesario desarrollar procesos sobre la plataforma GeneXus. Se debe intercalar en eventos de disparo específicos procesos para lograr exportar en forma de metadata los objetos del dominio necesarios para poder modificar reglas en el ambiente de ejecución.

En la sección 4.7 se detallan las operaciones de infraestructura agregadas al entorno de ejecución Genexus para generar metadata a utilizarse durante la edición de reglas en el ambiente de ejecución.

4.7. Generación de Metadata

Para poder hacer modificaciones sobre cualquier regla definida desde el IDE GeneXus en runtime, es necesario disponer de los siguientes componentes en el ambiente de ejecución:

- Objetos *Ruleset* y *RuleDSL*.
- Propiedades específicas de la plataforma.
- Hechos sobre los cuales se permiten definir reglas, en el contexto GeneXus serán *Business Components* y *SDTs*.
- Acciones que se pueden aplicar en las reglas, en este caso interesa conocer qué objetos GeneXus contiene la Base de Conocimiento.

Toda la información de contexto se lleva como XML a ejecución, mientras que las reglas y dependencias como archivos de texto.

A nivel de implementación, se utiliza el mecanismo de suscripción a eventos GeneXus para lograr la generación de la metadata necesaria.

En el evento de disparo *BeforeCompile* se genera a XML la información de contexto a usarse en runtime.

En el evento de disparo *AfterCompile* se copian todos los artefactos generados al directorio donde ejecuta la aplicación.

Se desarrolla un editor de reglas standalone (sección 4.8) para realizar modificaciones en runtime. Es importante resaltar que la edición de las reglas de negocio utiliza la misma representación GeneXus que se define desde el IDE, es decir, las modificaciones en el ambiente de ejecución se mantiene el enfoque independiente de la plataforma a nivel GeneXus (PIM).

4.7.1. Objetos Ruleset y RuleDSL

Para llevarse a tiempo de ejecución, cada *Ruleset* se transforma en un archivo de texto con extensión "gxrl" (*GeneXus Rule Language*) quedando disponible desde el contexto de ejecución, para que se puedan hacer modificaciones sobre el juego de reglas que contiene.

También se exportan las representaciones de *RuleDSL* existentes en la Base de Conocimiento para poder editar las reglas al nivel de abstracción del experto del *negocio*, cada objeto *RuleDSL* se corresponde con un archivo de texto en formato XML con extensión "*gxrdsl*" (*GeneXus Rule Domain Specific Language*).

4.7.2. Metadata GeneXus

Para poder editar las reglas en cualquiera de los lenguajes de especificación implementado, es necesario llevar a runtime metadata asociada con el modelo GeneXus que describa los diferentes objetos del dominio (los hechos), procedimientos, propiedades etc.

La metadata está compuesta de los siguientes archivos en formato XML [18]:

- rulesetbcs.xml
- rulesetdps.xml
- ruleseteds.xml
- rulesetenv.xml
- rulesetobjects.xml
- rulesetprocs.xml
- rulesetsdts.xml

En su conjunto, los archivos detallados anteriormente habilitan al experto del negocio a modificar el comportamiento de la aplicación a través de modificaciones sobre las reglas de negocio.

rulesetenv.xml

El archivo *rulesetenv.xml* detalla todas las propiedades del ambiente correspondiente al modelo GeneXus.

El siguiente ejemplo detalla alguna de las propiedades del modelo para los escenarios de uso plantedos:

Figura 4.11.: rulesetenv.xml: Propiedades del ambiente de ejecución

Es de particular interés propiedades como por ejemplo "*PACKAGE*" y "*DECIMAL_MATH*" que inciden en la generación de reglas de negocio.

La propiedad "*PACKAGE*" es necesaria para identificar objetos de negocio. En este caso, las clases Java que describan las diferentes entidades sobre las cuales se pueden definir reglas se califican según el valor de esta propiedad.

La propiedad "*DECIMAL_MATH*" describe si se debe de utilizar aritmética decimal[8] al realizar operaciones teniendo una incidencia directa en la forma de traducir las operaciones aritméticas en el lenguaje de reglas de la plataforma.

rulesetobjects.xml

El archivo *rulesetobjects.xml* detalla una lista de todos los objetos GeneXus que pueden intervenir en la definición de reglas de negocio.

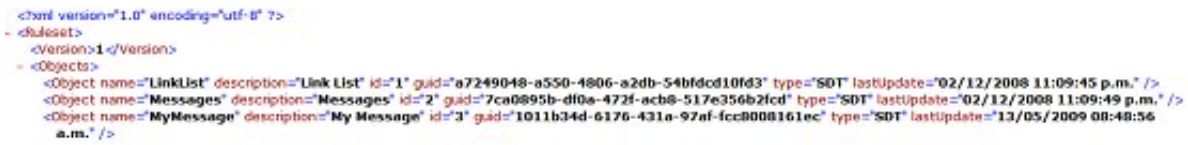


Figura 4.12.: rulesetobjects.xml: Objetos del dominio

A partir de este archivo se realizan las validaciones correspondientes cuando se salva un objeto *Ruleset* o *RulDSL*, asegurándose que los objetos de negocio sobre los cuales se definen reglas son correctos.

rulesetsdts.xml

El archivo *rulesetsdts.xml* detalla la composición de todos los objetos estructurados GeneXus, Structured Data Type:

[8] `http://java.sun.com/j2se/1.5.0/docs/api/java/math/BigDecimal.html`

Figura 4.13.: rulesetsdts.xml: Objetos de tipo SDT, propiedades y niveles

El archivo se utiliza para validar las propiedades de los objetos de negocio que se utilizan en el contexto de definición de consecuencias y acciones de una regla.

rulesetbcs.xml

El archivo *rulesetbcs.xml* detalla la composición de los *Business Components* que también son objetos estructurados con comportamiento extra predefinido.

Figura 4.14.: rulesetbcs.xml: Objetos de tipo BC, propiedades y niveles

Al igual que el caso anterior el archivo se utiliza para validar las propiedades de los objetos de negocio de tipo *Business Component* cuando se define una regla.

rulesetdps.xml

El archivo *rulesetdps.xml* detalla la composición de los objetos de tipo *DataProvider*:

```
<?xml version="1.0" encoding="utf-8" ?>
- <Ruleset>
    <Version>1</Version>
  - <DataProviders>
    + <DataProvider name="dpMessages" description="dp Messages" id="1" guid="6379e60a-218a-4b92-ab21-
        7e2085b3f47f" lastUpdate="04/07/2009 02:46:31 p.m.">
    + <DataProvider name="dpParameters01" description="dp Parameters01" id="2" guid="f96d73d1-27df-
        4ac3-a5a5-b3b216111555" lastUpdate="05/07/2009 08:38:04 a.m.">
    - <DataProvider name="TestExtendedCustomer" description="Test Extended Customer" id="3"
        guid="81cea9dd-cfa1-44eb-828a-d8da86dc0eec" lastUpdate="21/08/2009 12:11:52 a.m.">
      - <Signatures>
        - <Signature name="parm" validEnvironments="All">
          - <Parameters>
              <Parameter accessor="PARM_OUT" isAttribute="False" />
            </Parameters>
          </Signature>
        </Signatures>
      - <Properties>
          <Property name="Name" value="TestExtendedCustomer" isNull="false" />
          <Property name="Description" value="Test Extended Customer" isNull="false" />
```

Figura 4.15.: rulesetdps.xml: Objetos de tipo DataProvider, propiedades y estructura

ruleseteds.xml

El archivo *ruleseteds.xml* detalla los objetos dominio que a su vez son enumerados:

```
<?xml version="1.0" encoding="utf-8" ?>
- <Ruleset>
    <Version>1</Version>
  - <EDs>
      <ED name="RecentLinksOptions" description="Recent Links Options" id="1" guid="09d9015c-903a-4142-
        b78e-72dca73ba789" lastUpdate="02/12/2008 11:09:43 p.m." type="NUMERIC" length="4"
        decimals="0" signed="False" />
      <ED name="IMEMode" description="IMEMode property values" id="4" guid="f3df789c-6584-4ceb-9c6f-
        9e12fefb04ca" lastUpdate="02/12/2008 11:09:48 p.m." type="CHARACTER" length="40" decimals="0"
        signed="False" />
      <ED name="MessageTypes" description="Message Types" id="5" guid="60a17476-86c0-45f4-9405-
        bbe193690ed0" lastUpdate="02/12/2008 11:09:49 p.m." type="NUMERIC" length="2" decimals="0"
        signed="False" />
      <ED name="Gender" description="Gender" id="12" guid="4025405d-2382-4ad8-b360-49a0997550c6"
        lastUpdate="03/12/2008 06:37:31 a.m." type="CHARACTER" length="20" decimals="0"
        signed="False" />
      <ED name="MaritalStatus" description="Marital Status" id="13" guid="5304973f-6899-454e-9851-
        cf8fc50fe1fe" lastUpdate="03/12/2008 06:38:12 a.m." type="CHARACTER" length="20" decimals="0"
        signed="False" />
```

Figura 4.16.: ruleseteds.xml: Objetos de tipo Dominio Enumerado y sus propiedades

rulesetprocs.xml

El archivo *rulesetprocs.xml* detalla todos los procedimientos y sus firmas correspondientes:

Figura 4.17.: rulesetprocs.xml: Procedimientos y sus parámetros

El archivo se utiliza para validar que el uso de los procedimientos en el contexto de definición de consecuencias y acciones de una regla es válido.

4.8. Diseñador de Reglas en Ejecución

El editor de reglas de negocio para el entorno de ejecución es el componente más importante una vez que la aplicación se encuentra en producción. Utilizando dicho editor, el experto del negocio tiene acceso al juego de reglas que evalúa la aplicación, pudiendo hacer las modificaciones que sean necesarias para lograr la flexibilidad deseada en las aplicaciones.

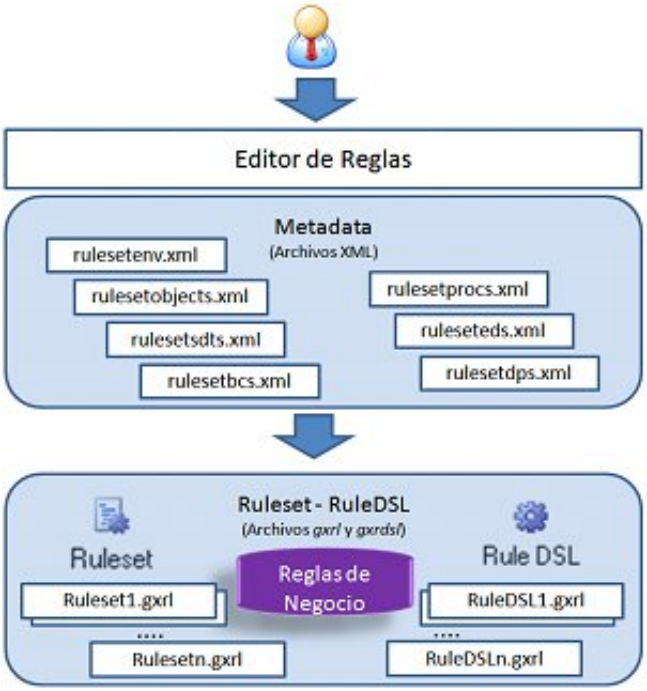


Figura 4.18.: Editor externo de reglas de negocio

Tal como detalla la Figura 4.18, a partir de la especificación de reglas de negocio y lenguajes de dominio en el directorio de ejecución de la aplicación representado en archivos de texto, se permiten hacer las modificaciones necesarias a las reglas dinámicas definidas a través de la metadata existente generada como XML con el editor de reglas.

Suponer que se tiene la aplicación de facturación de ejemplo (ver sección G.2) ejecutando en la siguiente ubicación:

http://localhost:8080/brsamplexev101ruleset01JavaEnvironment/servlet/

El caso de uso de segmentación de cliente (ver sección 5.2.2) ejecuta el servlet `com.sample.customertest01` en la URL que detalla la Figura 4.19.

Figura 4.19.: URL de ejecución de una aplicación de ejemplo

Se desea abrir el juego de reglas de la aplicación que está ejecutando. Se debe seleccionar la carpeta *Ruleset* debajo de la carpeta donde se encuentran los archivos relacionados con la aplicación web; que es la ubicación predeterminada de la metadata.

Figura 4.20.: Metadata de reglas en el directorio de ejecución

Se despliegan los archivos *gxrl* y *gxrdsl* asociados a los objetos *Ruleset* y *RuleDSL* existentes:

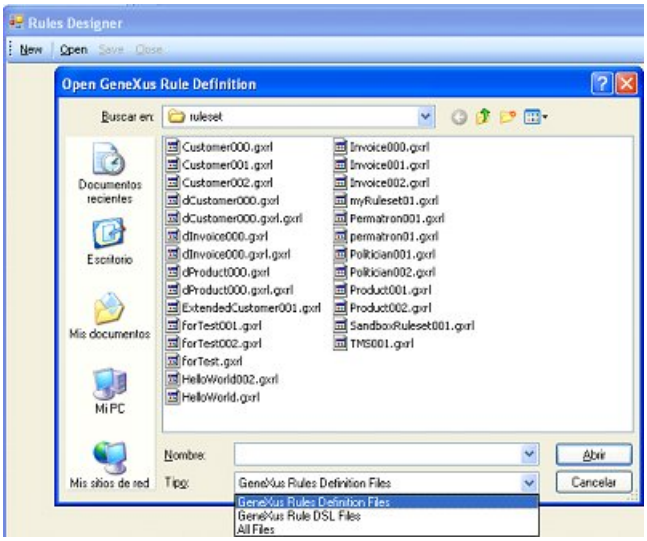


Figura 4.21.: Objetos Ruleset y RuleDSL en directorio de ejecución

Supongamos que se desea abrir el juego de reglas "*dProduct000*" que se corresponde con el escenario de uso del *Producto* (ver sección 5.2.3). Las reglas de negocio se despliegan en forma equivalente al IDE GeneXus, como se detalla en la Figura 4.22.

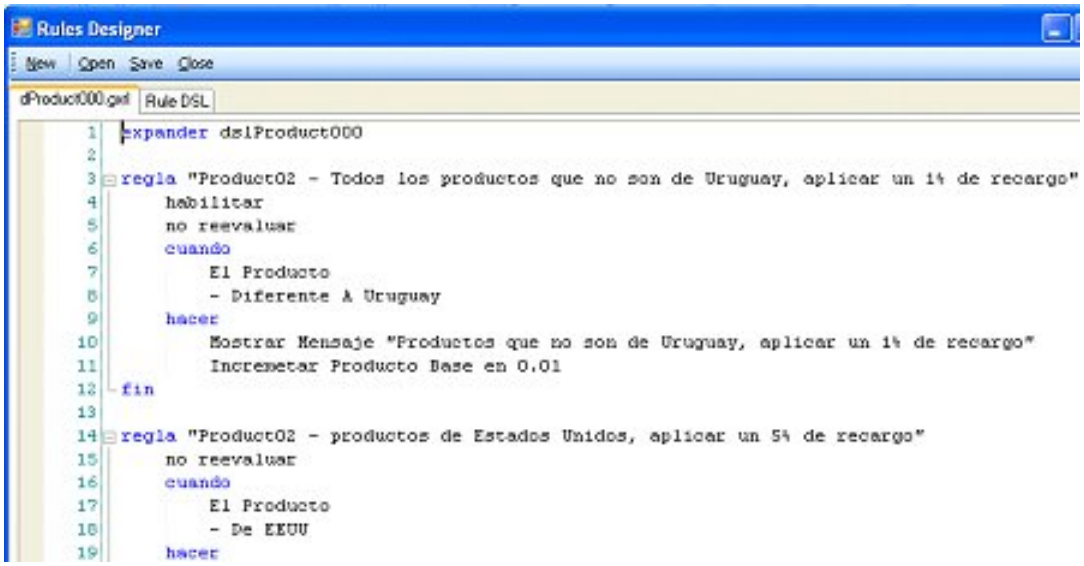


Figura 4.22.: Archivo de ejemplo de tipo Ruleset

En este caso está utilizando el objeto *RuleDSL* dslProduct000 (ver Figura 4.23) según detalla la palabra clave *expander* al inicio del juego de reglas:

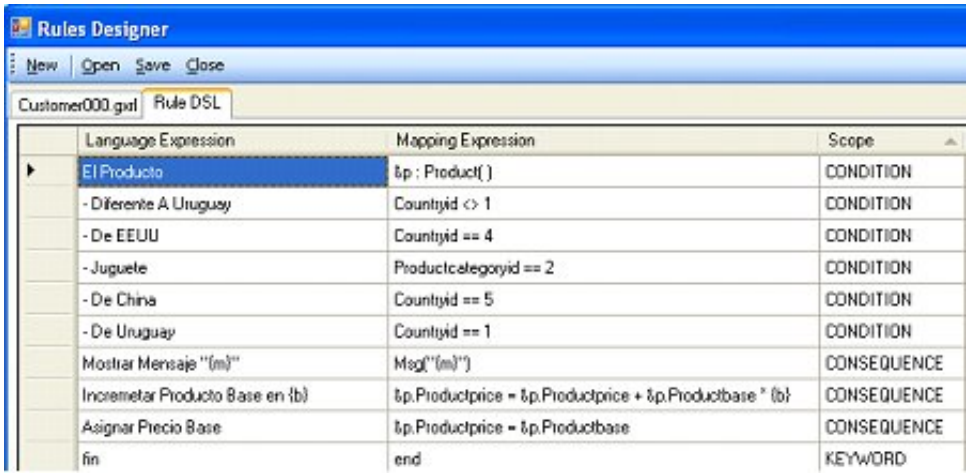


Figura 4.23.: Archivo de ejemplo de tipo RuleDSL

Utilizando dicho editor se permite realizar modificaciones a cualquier juego de reglas o crear nuevos juegos de reglas directamente en los entornos de ejecución. Al salvar un objeto de reglas, utilizando la metadata existente, se realiza la transformación del lenguaje de reglas de dominio a técnico (en caso de utilizar objetos *RuleDSL*) y posteriormente se transforma a la plataforma - en este caso el lenguaje DRL (Drools Rule Language) ya que se utiliza Drools. La próxima ejecución de la aplicación que involucre el juego de reglas salvado ejecutará la nueva definición de reglas.

Se implementan características básicas de *intellisense* utilizando *Ctrl+Space* para seleccionar las expresiones del lenguaje disponibles para manipular las reglas en la sección de condiciones o acciones de una regla.

Por ejemplo, a partir del lenguaje de alto nivel de la Figura 4.23 utilizando el acceso directo *Ctrl+Space* en la Figura 4.24 la sección de condiciones de cualquier regla se disponibilizan los bloques constructivos del lenguaje disponibles.

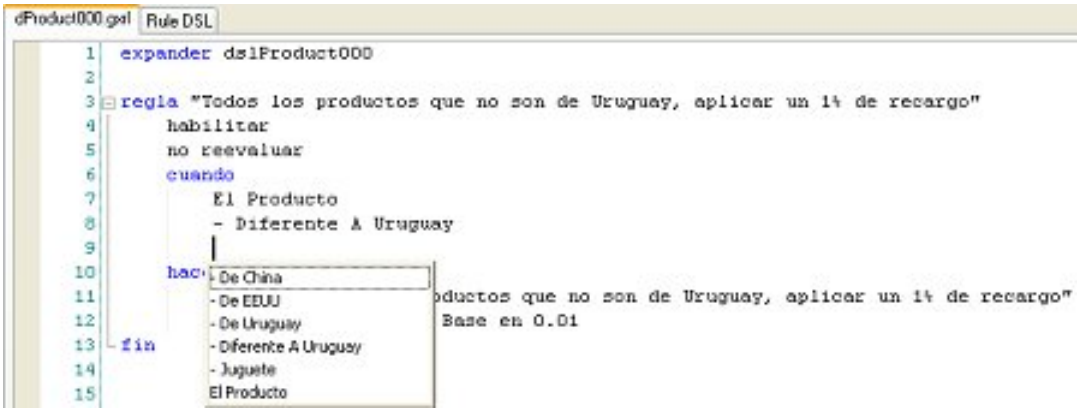


Figura 4.24.: Intellisense en sección de condiciones

En la Figura 4.25 la sección de consecuencias disponibiliza los bloques constructivos de tipo *CONSEQUENCE*, es decir las posibles acciones.

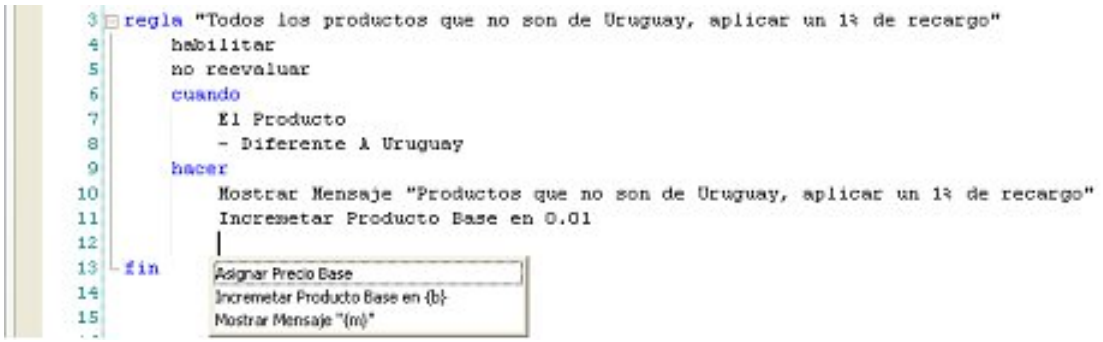


Figura 4.25.: Intellisense en sección de acciones

Notar que el experto del negocio, utilizando la capacidad de *intellisense* puede construir rápidamente un juego de reglas, y solo tiene que sustituir las variables (texto encerrado entre llaves "*{}*") con los valores que desee utilizar.

El diseñador de reglas standalone utiliza la misma lógica de negocio que se utiliza desde el IDE, y provee las mismas facilidades de edición de reglas. Internamente se cambia la lectura de los objetos del dominio sobre los cuales se construyen las reglas y la forma de salvado. La lectura de objetos utiliza los archivos XML generados (conceptualmente la metadata) y el salvado de objetos de reglas (objetos *RuleDSL* y *Ruleset*) se realizan en archivos de texto en vez de utilizar la Base de Conocimiento.

4.9. Traducción de Reglas a la plataforma

Al salvar un juego de reglas ya sea desde GeneXus como con el editor externo de reglas, se realiza la traducción del lenguaje GeneXus al lenguaje de la plataforma – en este caso a DRL (Drools Rule Language) utilizando Drools 5.0 [13] – y se almacena como un archivo de texto.

Volviendo al enfoque MDA (Model Driven Architecture) [191], en este punto se ejecutan las operaciones necesarias para convertir las reglas de negocio expresadas en forma independiente de la plataforma (*PIM*) a una representación dependiente de la misma (*PSM*) para que pueda ser ejecutada.

Antes de explicar como se implementó la generación de reglas en sí, se detallan los componentes principales que utiliza la plataforma GeneXus para generar objetos a partir de una Base de Conocimiento.

4.9.1. Generación de código GeneXus

Como detalla la sección 3.2.2, el generador utiliza la Base de Conocimiento y el archivo resultado del proceso de especificación para generar código en el lenguaje destino. La generación se hace en un solo paso, se utiliza Prolog para interpretar el archivo de especificación y teniendo en cuenta diferentes propiedades de la Base de Conocimiento realiza la transformación de esta *metadata* y se logra el fuente en la plataforma correspondiente. Posteriormente dependiendo de la plataforma puede ser necesario compilar los objetos para obtener una representación ejecutable de los mismos.

La figura 4.26 detalla gráficamente el proceso. A partir del objeto *Objeto_1*, el proceso de especificación obtiene el archivo *Objeto_1.spc*; el generador toma dicha definición y genera un objeto válido para la plataforma en este caso *Objeto_1.java* que posteriormente se compila y obtiene el archivo *Objeto_1.class*.

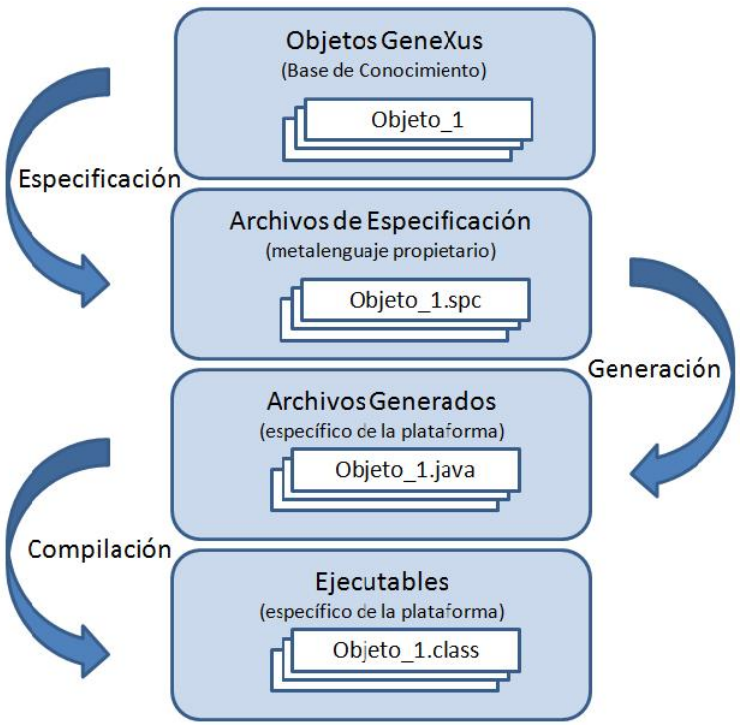


Figura 4.26.: Generación de código en GeneXus

4.9.2. Generación de Reglas

La incorporación del concepto de reglas dinámicas a través de objetos *Ruleset* implica traducir los bloques constructivos "*Condiciones*", "*Acciones*" y "*propiedades*" de cada regla (detallado en la sección 4.3).

La sección derecha (*acciones*) de una regla se podría generar en su totalidad por

el generador ya que se corresponde con código procedural GeneXus sobre los objetos del dominio ya existentes. Sin embargo, no es posible reutilizar al generador para traducir la sección izquierda (*condiciones*) y propiedades de una regla ya que son nuevas características. Para reutilizar el conocimiento del especificador y generadores sería necesario agregar nuevos predicados Prolog para permitir transformar en primera instancia las definiciones del nuevo objeto *Ruleset* en el metalenguaje que manipula el especificador y posteriormente extender el generador para que realice las transformaciones necesarias y obtener así la representación de la regla en la plataforma.

Además, es necesario modificar las reglas en el ambiente de ejecución (sección 4.8). Esto implica extender nuevamente al especificador y generadores agregando un mecanismo para que puedan interpretar los objetos del dominio a partir de la metadata generada para runtime (sección 4.7).

Se decide tomar un camino alternativo en la generación del código que representa a las reglas en la plataforma; no se reutilizan las rutinas estándar de especificación y generación e implementa un proceso de traducción independiente utilizando un árbol sintáctico (AST- Abstract Syntax Tree) [195] en base a las reglas de negocio de un objeto *Ruleset* que se utiliza desde el ambiente de desarrollo GeneXus (IDE) y editor externo de reglas.

La Figura 4.27 detalla los objetos sobre los cuales se trabaja en la plataforma de ejecución. Los objetos GeneXus generados están representados por los fuentes *Objeto_1.java* a *Objeto_n.java* y las reglas de negocio traducidas a la plataforma Drools a partir del lenguaje técnico GeneXus están representados por el lenguaje propietario *DRL* y se almacenan en los archivos de nombre *Ruleset_1.drl* a *Ruleset_n.drl*.

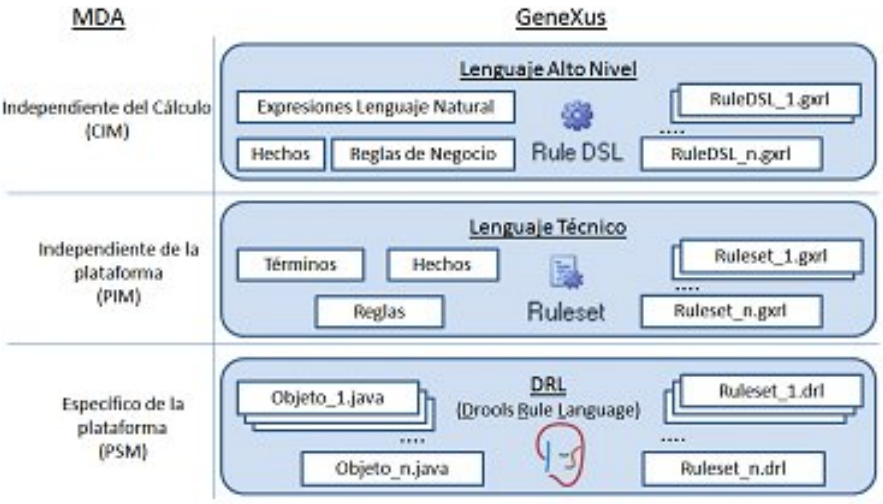


Figura 4.27.: Traducción de reglas a la plataforma

Se utiliza la herramienta Gold Parser (Grammar Oriented Language Developer) [27] versión 3.4.4 para definir la gramática del lenguaje técnico de reglas utilizando BNF (Backus-Naur Form) [180] y el parser Morozov Engine [26] en C# de tipo LALR (Lookahead Left to Right **P**arsing[9]) para validar la correctitud del código, es decir, análisis léxico, sintáctico y semántico.

Una vez creado y validado el análisis sintáctico, a medida que se realiza la validación semántica se recorre en forma top-down el árbol que representa el juego de reglas que se está salvando y se construye la representación DRL de las reglas que luego se almacena como archivo con extensión *drl* en el directorio de ejecución.

Se utiliza *StringTemplate*[10] (una biblioteca de plantillas utilizada para generar texto a partir de datos estructurados) para generar la representación DRL (Drools Rule Language) a partir de GXRL (GeneXus Rule Language).

4.9.3. Ejemplo de Traducción

Siguiendo con el escenario de uso del producto detallado en la sección 4.8, supongamos que se tiene la siguiente regla escrita en formato *RuleDSL* a partir del archivo "*dProduct000.gxrl*" de la Figura 4.22:

```
regla "Productos que no son de Uruguay, aplicar 1% de recargo"
  habilitar
  no reevaluar
  cuando
    El Producto
     - Diferente A Uruguay
  hacer
    Mostrar Mensaje "Se aplica un 1% de recargo"
    Incremetar Producto Base en 0.01
fin
```

Al salvar el objeto *Ruleset* y según el *RuleDSL* de nombre *dslProduct000*, se realiza la transformación al lenguaje técnico GeneXus (gxrl) y almacena en el archivo *dProduct000.gxrl.gxrl* (el algoritmo que realiza la transformación se detalla en la sección 4.4.4):

```
rule "Productos que no son de Uruguay, aplicar 1% de recargo"
  enabled false
  no-loop true
  when
    &p : Product( Countryid <> 1 )
  then
    Msg("Se aplica un 1% de recargo")
    &p.Productprice = &p.Productprice + &p.Productbase * 0.01
end
```

[9] http://www.devincook.com/goldparser/doc/about/lalr.htm
[10] http://www.antlr.org/wiki/display/ST/Five+minute+Introduction

A su vez se realiza la transformación al lenguaje destino DRL, obteniéndose la siguiente regla en el archivo *dProduct000.drl*:

```
package com.sample
rule "Productos que no son de Uruguay, aplicar 1% de recargo"
  enabled true
  no-loop true
  when
    $p:com.sample.SdtProduct(gxTv_SdtProduct_Countryid!=1)
  then
    com.ruleset.api.MessageHelper.msg("Se aplica un 1% de recargo");
    $p.setgxTv_SdtProduct_Productprice($p.getgxTv_SdtProduct_Productprice()
            .add($p.getgxTv_SdtProduct_Productbase()
            .multiply(new java.math.BigDecimal(0.01))));
end
```

Notar que en este caso se está utilizando toda la metadata generada en formato XML para obtener la representación en la plataforma. En este caso se especifica:

- Propiedad `Package` para indicar donde se encuentran los objetos del negocio. La generación de dicha propiedad está relacionada con las propiedades de ejecución en la plataforma y la propiedad GeneXus "`Package name`".
- La representación de objetos GeneXus se transforma a la representación de objetos Java generados; en este caso del objeto `Product` se transforma en `com.sample.sdtProduct` y la propiedad `CountryId` se convierte en `gxTv_SdtProduct_CountryId`.
- A su vez, la aritmética

  ```
  &p.Productprice = &p.Productprice + &p.Productbase * 0.10
  ```

  que representa la regla de producción que asigna el precio del producto, se convierte en la siguiente sentencia utilizando la semántica Java con el tipo de dato BigDecimal:

  ```
  $p.setgxTv_SdtProduct_Productprice(
  $p.getgxTv_SdtProduct_Productprice()
  .add($p.getgxTv_SdtProduct_Productbase()
  .multiply(new java.math.BigDecimal(0.01))));
  ```

  utilizando los correspondientes getter y setter según la especificación JavaBeans [38] para referenciar las propiedades del objeto `com.sample.SdtProduct`.

El archivo *dProduct000.drl* será el archivo utilizado en ejecución que contiene la especificación de reglas de negocio en la plataforma.

En términos generales, para cualquier archivo de reglas por ejemplo *<ArchivoDeReglas>.gxrl* si se utiliza un objeto *RuleDSL* primero se convierte dicha representación a formato técnico utilizando el algoritmo de Markov (sección 4.4.4), por lo que

se genera el archivo *<ArchivoDeReglas>.gxrl.gxrl*. A partir de esta representación utilizando el árbol sintáctico se genera un archivo con mismo nombre y extensión *drl* que será la traducción del juego de reglas en la propia plataforma de ejecución.

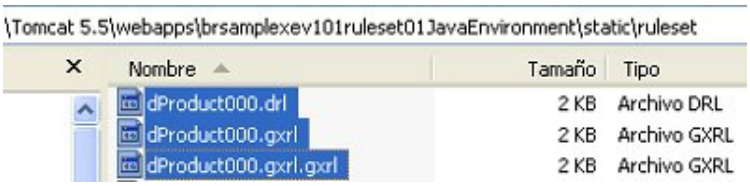


Figura 4.28.: Archivos de reglas traducidos en el directorio de ejecución

Los tres archivos generados detallan claramente los tres niveles de abstracción que detalla el enfoque MDA [191]:

- *CIM*: archivos de texto con extensión *gxrl* que utilizan un objeto *RuleDSL*; están utilizando el lenguaje de reglas de alto nivel de abstracción.
- *PIM*: archivos de texto con extensión *gxrl.gxrl*; representan reglas en formato técnico.
- *PSM*: archivos de texto con extensión *drl* que representan las reglas en la plataforma de ejecución.

4.9.4. Interacción con Drools 5.0

Para interactuar con el motor de evaluación a partir de los métodos que se exponen desde GeneXus (ver sección 4.5.1) se realiza un JAR externo (rulesetengine.jar) que centraliza el código utilizado para interactuar con el motor de reglas. Internamente el código generado GeneXus va a interactuar con el wrapper definido que a su vez va a utilizar los archivos *jar* que se corresponden con la herramienta Drools 5.0 para evaluar reglas dinámicas.

Notar que en este punto se evidencia los tres niveles de abstracción del modelo MDA [191]; se manipula el lenguaje de definición de reglas de dominio (nivel CIM) que utilizan los expertos del negocio. Se utilizan los objetos *Ruleset*, *RuleDSL* y la metadata GeneXus en formato XML para validar las reglas (nivel PIM). El nivel de la plataforma de ejecución (nivel PSM) implica la traducción de las reglas al lenguaje DRL para interactuar con Drools.

El proyecto Java *RulesetAPI* implementa dicha interfaz, las clases principales son las siguientes:

- *MessageHelper*: implementa el uso de la función GeneXus Msg (sección 3.4.5); para ello utiliza las clases estándar GeneXus Java en donde se implementa la función.

- *ObjectReferences*: almacena un diccionario con los objetos insertados en la memoria de trabajo; utilizado cuando se referencian las operaciones `insert`, `update` y `retract`.
- *PathHelper*: a partir de un string que representa un objeto *Ruleset*, se encarga de resolver el path completo en donde se encuentra la representación de reglas de negocio en la plataforma de ejecución.
- *RulesetEngine*: se centraliza la interacción con el motor de reglas en la plataforma. Ofrece métodos para crear sesiones con el motor de ejecución, insertar\modificar\eliminar objetos de la memoria de trabajo, disparar reglas y cerrar sesión.

4.10. Ciclo de ejecución de un juego de Reglas

Para explicar cómo funciona en ejecución la evaluación de reglas de negocio se toma como referencia el *Escenario 1 – Cliente* de la sección 5.2.2. El objetivo del ejemplo es segmentar el cliente en un tipo *A*, *B*, *C* utilizando un juego de reglas para que pueda ser modificable por un experto del negocio.

Se parte del diseño de la aplicación en el ambiente de desarrollo GeneXus. El WebPanel *customertest01* de la Figura 4.29 será el objeto con el cual el usuario final interactúa.

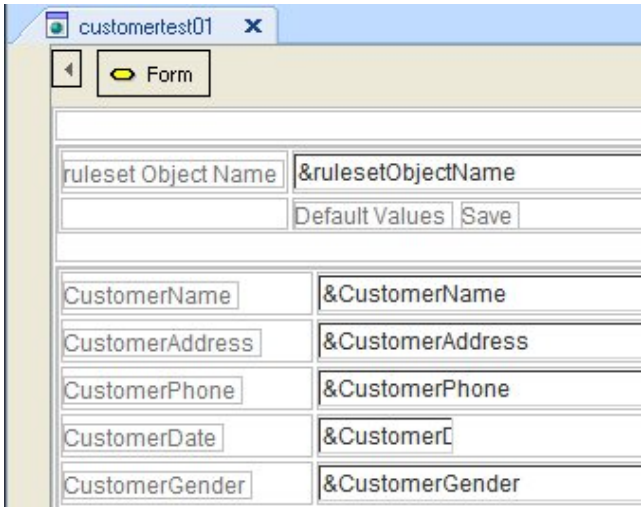


Figura 4.29.: Objeto de ejemplo para crear Clientes

La opción *Save* se encarga de salvar el cliente, pero antes evalúa un juego de reglas específico. El evento que se ejecuta al presionar el botón *Save* es el siguiente:

```
Sub 'OnSavingCustomer'
  &myCustomer = new()
  &myCustomer.CustomerId = &CustomerId
  &myCustomer.CustomerName = &CustomerName
  &myCustomer.CustomerAddress = &CustomerAddress
  ...
  Do 'InteractWithRuleEngine'
  &myCustomer.Save()
  If &myCustomer.Success()
    &sdtToXML = "Insert OK"
    commit
  else
    &sdtToXML = "Insert Error:" + &myCustomer.GetMessages().ToXml()
  EndIf
EndSub
```

Notar que se crea un objeto de tipo `Customer` y se asignan todas las variables en pantalla al objeto. En este punto se define el evento de disparo donde se validan las reglas de negocio; recordando el patrón Event-Condition-Action [118] queda explícito dentro del código procedural de la subrutina `InteractWithRuleEngine`.

La interacción con el motor de evaluación de reglas se define de la siguiente forma:

```
// "RulesetObjectName"
&myEngine.Ruleset(&myRulesetObject)

// insert object to the Engine
&ret = &myEngine.Insert(&myCustomer)
&myEngine.FireRules()
&myEngine.Close()
```

La variable `&myEngine` representa un objeto de tipo *RulesetEngine* que es efectivamente el motor de reglas. Se le indica el juego de reglas a utilizar mediante la variable `&myRulesetObject`. Es importante resaltar que el dinamismo en la evaluación de reglas y políticas de negocio se logra al poder agregar, eliminar o modificar lógica de negocio en un evento de disparo específico a través de un juego de reglas modificable en tiempo de ejecución.

En este caso al referenciar en una variable al juego de reglas a utilizarse, no sólo permite modificar cualquier regla contenida en este grupo sino que también se puede parametrizar para utilizar diferentes juegos de reglas. La modificación de las reglas de negocio en el ambiente de ejecución se realiza modificando los objetos *Ruleset* y *RuleDSL* a través del diseñador de reglas en el ambiente de ejecución (sección 4.8).

La inserción en la memoria de trabajo de los objetos sobre los cuales aplican reglas se realiza a través del método `Insert`; en el ejemplo analizado se inserta sólo el cliente que se desea validar a través de la variable `&myCustomer`. En caso que la evaluación de reglas implique más objetos, se deberá utilizar repetidamente el método `Insert`.

Posteriormente, se ejecuta el método `FireRules`. Es durante la ejecución del método `FireRules` donde se manifiestan las secciones condición-acción del patrón Event-Condition-Action [118]. Para cada regla que cumpla las condiciones, el motor de reglas se encarga de ejecutar las acciones correspondientes.

Finalmente, se cierra la sesión con el motor de reglas, se devuelve el control al objeto GeneXus llamador y continúa la ejecución del programa.

Generalizando, la Figura 4.30 detalla las diferentes etapas que implica desarrollar una aplicación que utilice el esquema de evaluación dinámica de reglas de negocio, desde el diseño de la aplicación utilizando los objetos GeneXus teniendo en cuenta el patrón Event-Condition-Action hasta la ejecución correspondiente detallando las herramientas de las que dispone el experto del negocio para influir en forma directa en el comportamiento de la aplicación.

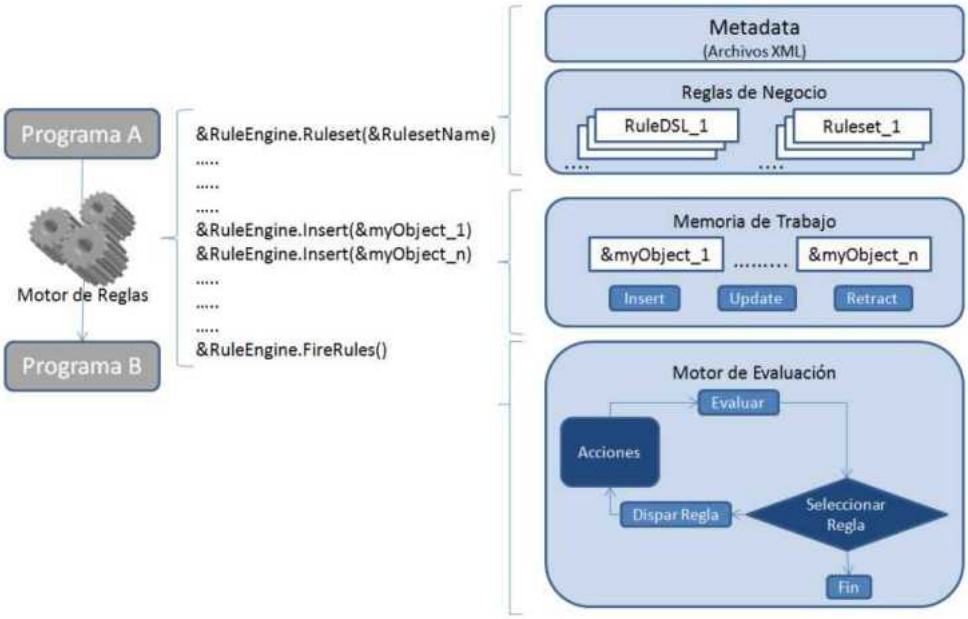


Figura 4.30.: Ciclo de desarrollo utilizando el enfoque "BRA"

Para desarrollar una aplicación que utilice reglas de negocio dinámicas en el entorno de ejecución GeneXus se debe tener en cuenta lo siguiente:

- *Infraestructura*: El desarrollador GeneXus se encarga de crear las reglas iniciales e infraestructura base necesaria en la base de conocimiento correspondiente.
- *Evento de disparo*: Planificar en el código de la aplicación los eventos de disparo en los cuales se quiere tener capacidad de evaluar en forma dinámica reglas. Esto implica agregar código procedural para ejecutar en forma explícita al motor de reglas; en el ejemplo referirse al código de la rutina `InteractWithRuleEngine`.

- *Contexto*: Insertar en la memoria de trabajo del motor los diferentes objetos sobre los cuales aplican las reglas.
- *Parametrización del lenguaje de reglas*: Para que las reglas de negocio sean entendidas por el experto del negocio, definir al menos un lenguaje de "alto nivel" de abstracción en un objeto de tipo `RuleDSL` y definir las reglas (objetos de tipo `Ruleset`) utilizando dicho lenguaje.
- *Conexión Aplicación - Motor de reglas*: Asignar un juego de reglas al motor de evaluación de reglas utilizando el método `Ruleset` en el evento de disparo correspondiente.
- *Disparo de reglas*: Especificar el disparo de reglas en forma explícita al utilizar el método `FireRules` del objeto *RulesetEngine*.

Si se tienen en cuenta las consideraciones detalladas anteriormente, se logra separar lógica de negocio clave de la aplicación para que sea manipulada usando reglas de negocio.

El editor externo de reglas (sección 4.8) utilizando la metadata GeneXus correspondiente (sección 4.7.2) permite realizar modificaciones y agregar nuevos juegos de reglas, impactando en forma directa en la ejecución de la aplicación que las utiliza.

4.11. Prototipo

Como objetivo principal del prototipo, se persigue la validación de los conceptos planteados en el capítulo 3. Se experimenta con las ideas que propone el enfoque de uso de reglas de negocio y los motores de evaluación de reglas en el contexto GeneXus.

Como ya se mencionó, la representación en forma abstracta e independiente de la plataforma de las reglas de negocio se logra con la implementación de dos objetos nuevos dentro del contexto de ejecución GeneXus denominados *Ruleset* y *RuleDSL*. La evaluación de dichas reglas se resuelve encapsulando en un objeto externo GeneXus denominado *RulesetEngine*, las funcionalidades principales del motor de evaluación.

Se permite separar del código procedural aquellas restricciones y políticas de negocio que sean críticas por su alta volatilidad, de tal forma que si es necesario realizar cambios en las mismas, un experto del negocio utilizando el editor de reglas (detallado en la sección 4.8) pueda hacer las modificaciones sin necesidad de reprogramar la aplicación. Esta es la clave para lograr aplicaciones dinámicas con alta flexibilidad y adaptables al cambio.

Específicamente se plantea como objetivo poder cerrar un ciclo de desarrollo de una aplicación que utilice el concepto de reglas de negocio dinámicas. Esto significa

que se debe resolver desde su concepción, diseño de las reglas de negocio identificando sus términos, generación de código y metadata necesaria para que la aplicación pueda evaluar las reglas definidas y finalmente realizar modificaciones en tiempo de ejecución para validar el enfoque.

Se desarrolla una solución Visual Studio 2008 que representa la solución completa para administrar reglas de negocio dinámicas en la plataforma GeneXus, consta de los siguientes proyectos:

- *RulesetBL*: centraliza la lógica de negocio de la extensión.
- *RulesetUI*: interfaz de usuario dentro del contexto GeneXus.
- *RulesetEditor*: editor externo de reglas para utilizarse en el entorno de ejecución.
- *RulesetCommon*: lógica común reutilizada en todos los proyectos.

4.11.1. RulesetBL

Representa la lógica de negocio (Business Logic) de la extensión, desde la cual se genera la metadata necesaria para poder definir reglas de negocio en tiempo de ejecución. También se implementa toda la infraestructura para copiar en forma transparente todas las dependencias y artefactos relacionados cuando se generan los demás objetos GeneXus.

Se suscribe a los siguientes eventos de disparo GeneXus:

BeforeCompile:

- Se verifica la existencia de todas las dependencias a nivel de plataforma y se copian al directorio de generación y ejecución (rutina *CopyDrivers*).
- Se recorren los objetos GeneXus y genera la metadata correspondiente copiándose la representación XML a los directorios de generación y ejecución (rutina *GenerateRulesetMetadata*).

OnAfterCompile:

- Se copian todos los archivos de reglas al directorio de ejecución (rutina *CopyRulesetFiles*).

4.11.2. RulesetUI

Define nuevos objetos GeneXus y sus partes, que se corresponden con la representación de reglas; esto es, el objeto *Ruleset* y el lenguaje de alto nivel de declaración de reglas *RuleDSL*.

4.11.3. RulesetEditor

Editor externo de reglas utilizado para modificar las reglas de negocio en tiempo de ejecución.

4.11.4. RulesetCommon

Contiene toda la infraestructura común a los demás proyectos, entre otras cosas:

- Define la gramática del lenguaje técnico de reglas, su validación léxica y representación sintáctica.
- Define las rutinas de traducción de lenguaje de dominio a lenguaje técnico.

4.12. Consideraciones

Se detalla a continuación diferentes consideraciones realizadas durante la implementación del prototipo.

4.12.1. Lenguaje de Reglas

Recordando la propuesta para representar reglas de negocio que se definió a partir de la sección 3.5, varias concesiones se realizaron con respecto a algunos operadores y operaciones planteadas.

El operador *eval* permite ejecutar cualquier código procedural en la sección de condiciones de una regla, donde el resultado de la evaluación debe retornar un tipo de dato boolean. El operador no se implementa en su totalidad; solo se soporta comparación de constantes en vez de utilizar una expresión arbitraria como argumento.

La operación *AddError* del motor de evaluación no se pudo implementar utilizando la infraestructura ya existente a nivel de código generado. Cuando se trabaja con objetos GeneXus de tipo *Business Component*, internamente existe una propiedad *anyError* que se maneja a nivel de código generado que es actualizada en forma automática por el generador GeneXus para conocer si hay errores en el proceso que dicho objeto ejecuta cuando se intenta actualizar. Inicialmente se utiliza el mismo mecanismo desde la evaluación dinámica de reglas en el contexto de ejecución del motor de reglas para informar al objeto de la existencia de ún error. El problema encontrado es que el generador inicializa en forma automática dicha variable por lo que se pierden los cambios realizados por el motor de reglas.

Durante el proceso de investigación de cómo resolver la ejecución de objetos procedimientos ya existentes en la base de conocimiento como forma de reutilizar la

lógica de negocio en el contexto de reglas de negocio, se detectó que el generador agrega código extra cuando el pasaje de parámetros entre procedimientos es de tipo *Out* o *InOut*[11]. Se implementa el pasaje de parámetros de *In* (entrada) quedando fuera el realizar pruebas de ejecución sobre objetos de tipo DataProvider, ya que en forma predeterminada el pasaje de parámetros con dicho objeto tiene que ser de tipo *InOut*.

Durante el proceso de investigación utilizando lenguajes de *alto nivel* se experimenta con el desarrollo de lenguajes de reglas que utilicen caracteres doble byte (DBCS[12]). Los conjuntos de caracteres doble byte permiten al software mostrar y procesar datos en caracteres ideográficos. Los idiomas ideográficos incluyen el japonés, el coreano y el chino (simplificado y tradicional). Se detecta que no es posible realizar la especificación de reglas que utilicen caracteres *doble-byte* a nivel de la gramática desarrollada, no se investigaron posibles causas y alternativas para solucionar el problema a nivel del producto utilizado *Gold Parser* [27].

La comparación de valores y asignación en el lenguaje GeneXus utiliza el operador igual "=". En el desarrollo de la gramática para el lenguaje de reglas técnico, se selecciona el operador "==" para utilizarse en la sección izquierda (LHS) de una regla. Se busca dejar explícito el concepto de comparación de condiciones que implica la sección izquierda de una regla para que no se confunda con el operador de asignación "=".

4.12.2. Edición de Reglas

En la sección 4.8 se detallan las características principales del editor de reglas. Dos recursos importantes que ayudarían en la edición de reglas de negocio mediante el mecanismo de *intellisense* es tener la posibilidad de acceder a la base de datos y utilizar la infraestructura existente a nivel de dominios enumerados[13] de la Base de Conocimiento GeneXus para facilitar la definición de restricciones sobre los objetos del dominio. Durante el proceso de construcción de la metadata necesaria para editar reglas en el ambiente de ejecución se tuvieron en cuenta dichas consideraciones generándose la infraestructura necesaria para soportar la funcionalidad, sin embargo no se realizó una implementación que explote dichas características.

4.12.3. Plataforma de Generación

Inicialmente, el objetivo planteado involucra el desarrollo de un prototipo completo para las principales plataformas de ejecución GeneXus, es decir, brindar una solución de administración de reglas de negocio dinámica completa para los lenguajes de generación Java, C# y Ruby.

[11] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?8220`
[12] `http://www.computeruser.com/dictionary/double-byte-character-set/`
[13] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?9918`

Se realiza un análisis de herramientas existentes en el mercado tanto comerciales como open source (detallado en el Anexo E) y se analiza al menos una versión para Java, C# y Ruby investigando posibles mecanismos de integración con GeneXus utilizando código generado. Se seleccionan las herramientas Drools [13] para la plataforma Java, Drools.Net [14] para C# y Ruleby [85] para Ruby.

Se decide comenzar el desarrollo del prototipo con la plataforma de ejecución Java debido a que la selección de herramientas realizada es la más utilizada dentro del contexto open-source ajustándose a los intereses del proyecto.

Debido a atrasos en el desarrollo del prototipo en el entorno Java se decide dar prioridad al lenguaje de definición de reglas de alto nivel y lograr realizar un ciclo completo de prototipación utilizando el generador GeneXus Java por lo que no se implementa la integración de los motores de evaluación de reglas para los ambientes C# y Ruby según investigación realizada a nivel de código generado (Anexo E).

5. Escenarios de Uso

En este capítulo se realiza una revisión de los escenarios más importantes analizados en el Anexo G, utilizados para evaluar el prototipo construido.

Inicialmente se detalla el negocio sobre el cual se construye el ejemplo. Posteriormente se explicitan los casos en donde se desea tener la posibilidad de agregar\modificar comportamiento en tiempo de ejecución y como aplica una solución utilizando reglas de negocio.

Para cada ejemplo se especifica el juego de reglas que se desea evaluar. Se crean las reglas en el lenguaje de especificación de reglas de negocio en formato técnico y donde aplique, se detalla el formato "alto nivel" orientado al experto del negocio. Se agrega el código necesario para poder evaluar los juegos de reglas y por último se resume la evaluación de cada escenario detallando cómo se resuelve el problema.

5.1. Descripción del negocio

El ejemplo canónico se basa en una pequeña aplicación de compras de una empresa. Las entidades principales son Cliente (Customer), Tipo De Cliente (CustomerType), Producto (Product), Categoría De Producto (ProductCategory) y Factura (Invoice) como muestra la Figura 5.1:

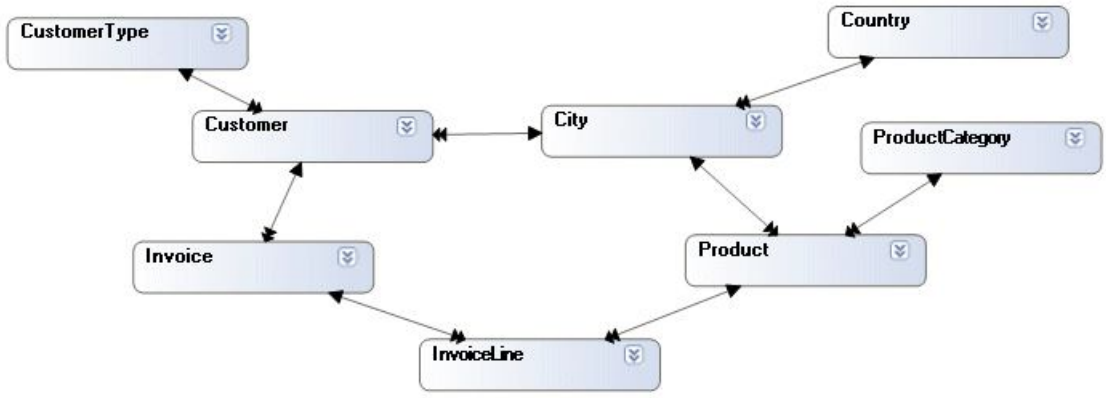


Figura 5.1.: Ejemplo canónico

Las operaciones fundamentales que ocurren son:

- Registro de Clientes.

- Ingreso y categorización de Productos.
- Registro de Facturas.

Cada entidad definida anteriormente es representada en el ecosistema GeneXus por un objeto de tipo *Business Component*.

Se parte de una aplicación GeneXus en donde se implementa la persistencia de los diferentes objetos del negocio sin indicar ningún tipo de comportamiento extra, y se define una interfaz de usuario básica para poder interactuar con las entidades.

Se pretende agregar comportamiento según sea necesario, sin modificar el código de la aplicación, utilizando reglas de negocio. Una breve descripción sobre los casos de uso se detalla a continuación:

- *Cliente*: segmentación y asignación del máximo de compra permitido por cliente.
- *Producto*: inicialización del precio de un producto utilizando diferentes consideraciones como por ejemplo, categoría y procedencia.
- *Factura*: aplicar diferentes tipos de validaciones, descuentos y promociones cuando se generan facturas.
- *Workflow*: separar ciertas decisiones de un flujo de negocio utilizando reglas.
- *OAV* y *modelos universales de datos*[1]: a partir de modelos genéricos que se utilizan para representar cualquier tipo de Entidad, agregar comportamiento y validaciones sobre las entidades, sus propiedades y relaciones.
- *Características Avanzadas*: experimentación con características avanzadas de los motores de evaluación como por ejemplo inferencia y *Truth Maintenance* (inserción lógica) [142, 207].

5.2. Escenarios

Los escenarios que se detallan a continuación aumentan en forma paulatina su complejidad. En esta sección se presentan con mayor profundidad aquellos ejemplos que resuelven diferentes tipos de problemas utilizando la evaluación de reglas de negocio. Por el detalle de todos los escenarios evaluados referirse al Anexo G.

- *Escenario 0 – Hola Mundo*: caso base para comprender el funcionamiento del motor de evaluación de reglas en el contexto GeneXus.
- *Escenario 1 – Cliente*: utilizando el ejemplo canónico detallado en la sección 5.1 se definen reglas de negocio sobre el ingreso y actualización de clientes.

[1] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?12273`

- *Escenario 4 – Workflow*: se explora el uso de reglas de negocio integrado en procesos de negocio.

- *Escenario 5 – OAV*: se experimenta la inclusión de reglas de negocio en modelos de datos genéricos.

- *Escenario 6 – Interacción con código GeneXus*: se investiga el uso de objetos GeneXus desde la evaluación dinámica de reglas.

- *Escenario 9 – "Truth Maintenance"*: se valida el uso del motor de evaluación de reglas como mecanismo de inferencia o razonamiento.

5.2.1. Escenario 0 – Hola Mundo

El escenario *Hola Mundo* consiste en la prueba inicial a realizar para evaluar el funcionamiento del motor de reglas, entender cómo se debe instanciar el motor de evaluación y asignar el juego de reglas y objetos sobre los cuales se quiere aplicar las reglas.

En términos generales, el funcionamiento es el siguiente:

- Definición del juego de reglas utilizando objetos de tipo *Ruleset* y *RuleDSL* en el IDE GeneXus o en el editor externo de reglas (sección 4.8).

- Desarrollar código procedural para interactuar con el motor de evaluación de reglas (sección 4.10), que incluye:

  - Instanciación del motor de evaluación de reglas.
  - Asignación de un juego de reglas.
  - Inserción de los objetos de dominio en la memoria de trabajo.
  - Ejecución del método que evalúa las reglas (método *FireAllRules* del objeto *RulesetEngine*).

El resultado de la evaluación de reglas fue satisfactorio y el detalle se encuentra como ya se mencionó en el Anexo G.

Se valida la ejecución hacia adelante del motor de reglas (mecanismo de inferencia *forward chaining*), la habilidad del motor para evaluar, agendar, activar y disparar en secuencia las diferentes consecuencias de las reglas, y también se considera los cambios que se realizan a los objetos durante ésta ejecución.

5.2.2. Escenario 1 – Cliente

El escenario relacionado con el mantenimiento de clientes tiene como objetivo poder realizar modificaciones en términos de restricciones y validaciones una vez que la aplicación ya se encuentra en ejecución. El cliente está representado por dos objetos de negocio denominados `Customer` y `CustomerType` como detalla la Figura 5.2:

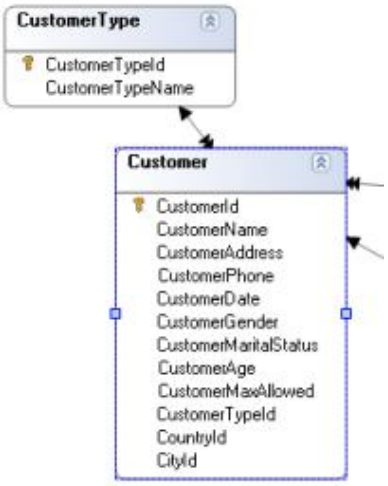


Figura 5.2.: Objeto Cliente y Tipo de Cliente

Un cliente (`Customer`) está formado por las siguientes propiedades:

- *CustomerId*: Identificador.
- *CustomerName*: Nombre.
- *CustomerAddress*: Dirección.
- *CustomerPhone*: Teléfono.
- *CustomerDate*: Fecha de nacimiento.
- *CustomerGender*: Género.
- *CustomerMaritalStatus*: Estado matrimonial.
- *CustomerAge*: Edad.
- *CustomerMaxAllowed*: Máximo de compras permitido.

A su vez, el mismo tiene asociado un tipo de cliente (*CustomerType*):

- *CustomerTypeId*: Identificador.

- *CustomerTypeName*: Descripción del tipo de cliente.

Los tipos de clientes pueden ser A, B o C referenciados por los identificadores 1, 2 y 3 respectivamente. Se le asigna *Ciudad* y *País* de Origen:

- *CityId*: Identificador de ciudad.
- *CountryId*: Identificador de país.

Inicialización

En tiempo de actualización del cliente se desea segmentarlo en los tipos A, B, C utilizando las siguientes reglas y también se desea poder modificar fácilmente dichas definiciones en el futuro.

- *Regla 1*: Todos los clientes por defecto son de tipo B.

```
rule "Todos los clientes son B"
  no-loop true
  when
    &c : Customer(CustomerTypeId=="")
  then
    Msg("Todos los clientes son B")
    &c.CustomerTypeId = 2 // B
end
```

- *Regla 2*: Clientes masculinos de hasta 25 años son de tipo A.

```
rule "Clientes masculinos de hasta 25 años"
  no-loop true
  when
    &c : Customer( Customergender == "M" ,
                   Customerage <= 25 )
  then
    Msg("Clientes masculinos de hasta 25 años son de tipo A.")
    &c.Customertypeid = 1 // A
end
```

- *Regla 3*: Clientes femeninos de hasta 30 años son de tipo A.

```
rule "Clientes femeninos de hasta 30 años"
  no-loop true
  when
    &c : Customer( Customergender == "F",
                   Customerage <= 30 )
  then
    Msg("Clientes femeninos de hasta 30 son de tipo A.")
    &c.Customertypeid = 1 // A
end
```

- *Regla 4*: Clientes de más de 70 años son de tipo C.

```
rule "Clientes de mas de 70 años"
  no-loop true
  when
    &c : Customer( Customerage > 70 )
  then
    Msg("Clientes de mas de 70 son de tipo C.")
    &c.Customertypeid = 3 // C
end
```

Modificaciones

Debido a cambios en la realidad se desea aplicar los siguientes cambios.

Disminuir la franja etárea que segmenta a los clientes masculinos de tipo A de 30 a 20 años. Se modifica la Regla 2 de la siguiente manera:

- *Regla 2'*: Clientes masculinos de hasta 20 años son de tipo A.

```
rule "Clientes masculinos de hasta 20 años"
  no-loop true
  when
    &c : Customer( Customergender == "M", Customerage <= 20 )
  then
    Msg("Clientes masculinos de hasta 20 son de tipo A.")
    &c.Customertypeid = 1 // A
end
```

Agregar una nueva segmentación: aquellos Clientes entre 50 y 60 años pasan a pertenecer al tipo C; se agrega una nueva regla:

- *Regla 5*: Clientes entre 50 y 60 años asignar el tipo C.

```
rule "Clientes entre 50 y 60 años"
  no-loop true
  when
    &c : Customer( Customerage >= 50 , Customerage <= 60 )
  then
    Msg("Clientes entre 50 y 60 asignar el tipo C.")
    &c.Customertypeid = 3 // C
    update(&c)
end
```

Crear nuevas restricciones sobre otros términos del dominio. Para aquellos clientes que sean de tipo C asegurarse que el máximo permitido de compra sea 300. Se agrega la siguiente regla:

- *Regla 6*: Clientes de tipo C asignar 300 como máximo de compras.

```
rule "Clientes C tiene un máximo definido de compras"
  enabled true
  no-loop true
  when
    &c : Customer( Customertypeid == 3 )
  then
    Msg("Clientes de tipo C asignar 300 como máximo de compras.")
    &c.Customermaxallowed = 300.00
end
```

Evaluación

Se crea un procedimiento de ejemplo para dar de alta dos clientes con los siguientes datos iniciales:

```
Sub 'variables01'  // cliente femenino de 30 años
  &CustomerName = "Customer " + &CustomerId.ToString().Trim()
  &CustomerAddress = "Address " + &CustomerId.ToString().Trim()
  &CustomerAge = 30
  &CustomerDate = YMDtoD( 1978,9,7 )
  &CustomerGender = Gender.Female
  &CustomerMaritalStatus = MaritalStatus.Single
  &CustomerPhone = "9009090"
  &CustomerMaxAllowed = 100
  &CustomerTypeId = 1
  &CountryId = 1
  &CityId = 1
EndSub

Sub 'variables02'
  // cliente masculino de 55 años
  &CustomerName = "Customer " + &CustomerId.ToString().Trim()
  &CustomerAddress = "Address " + &CustomerId.ToString().Trim()
  &CustomerAge = 55
  &CustomerDate = YMDtoD( 1953,9,7 )
  &CustomerGender = Gender.Male
  &CustomerMaritalStatus = MaritalStatus.Single
  &CustomerPhone = "9001010"
  &CustomerMaxAllowed = 99
  &CustomerTypeId = 1
  &CountryId = 1
  &CityId = 1
EndSub
```

Se ejecuta el escenario utilizando los dos juegos de reglas definidos inicialmente con las reglas que se detallan en la sección de inicialización, obteniendo el siguiente resultado sobre el primer cliente:

```
Clientes femeninos de hasta 30 son de tipo A.
```

El estado del cliente se detalla a continuación asignándose el tipo de cliente *A*:

```
<Customer xmlns="brsamplexev101">
  <CustomerId>152</CustomerId>
  <CustomerName>Customer 151</CustomerName>
  <CustomerAddress>Address 151</CustomerAddress>
  <CustomerPhone>9009090</CustomerPhone>
  <CustomerDate>1978-09-07</CustomerDate>
  <CustomerGender>F</CustomerGender>
  <CustomerMaritalStatus>S</CustomerMaritalStatus>
  <CustomerAge>30</CustomerAge>
  <CustomerMaxAllowed>100.00</CustomerMaxAllowed>
  <CustomerTypeId>1</CustomerTypeId>
  <CustomerTypeName>A</CustomerTypeName>
  <CountryId>1</CountryId>
  <CityId>1</CityId>
</Customer>
```

Para el segundo cliente no se ejecutan reglas:

```
<Customer xmlns="brsamplexev101">
  <CustomerId>153</CustomerId>
  <CustomerName>Customer 152</CustomerName>
  <CustomerAddress>Address 152</CustomerAddress>
  <CustomerPhone>9001010</CustomerPhone>
  <CustomerDate>1953-09-07</CustomerDate>
  <CustomerGender>M</CustomerGender>
  <CustomerMaritalStatus>S</CustomerMaritalStatus>
  <CustomerAge>55</CustomerAge>
  <CustomerMaxAllowed>99.00</CustomerMaxAllowed>
  <CustomerTypeId>1</CustomerTypeId>
  <CustomerTypeName>A</CustomerTypeName>
  <CountryId>1</CountryId>
  <CityId>1</CityId>
</Customer>
```

Sin modificar el juego de datos, se hacen las modificaciones sobre las reglas que detalla la sección *Modificaciones* y ejecuta nuevamente. Para el primer cliente se ejecutan exactamente las mismas reglas mientras que para el segundo cliente aplican las nuevas restricciones:

```
Clientes entre 50 y 60 asignar el tipo C.
Clientes de tipo C asignar 300 como máximo de compras.
```

El segundo cliente al finalizar la evaluación de reglas de negocio tiene el siguiente estado:

```
<Customer xmlns="brsamplexev101">
  <CustomerId>155</CustomerId>
  <CustomerName>Customer 154</CustomerName>
```

```
  <CustomerAddress>Address 154</CustomerAddress>
  <CustomerPhone>9001010</CustomerPhone>
  <CustomerDate>1953-09-07</CustomerDate>
  <CustomerGender>M</CustomerGender>
  <CustomerMaritalStatus>S</CustomerMaritalStatus>
  <CustomerAge>55</CustomerAge>
  <CustomerMaxAllowed>300.00</CustomerMaxAllowed>
  <CustomerTypeId>3</CustomerTypeId>
  <CustomerTypeName>C</CustomerTypeName>
  <CountryId>1</CountryId>
  <CityId>1</CityId>
</Customer>
```

Lenguaje de Alto nivel

Utilizando el segundo juego de reglas, se desarrolla un lenguaje de alto nivel de abstracción en formato inglés para experimentar con una definición de reglas de negocio acorde al experto del negocio. Para ello se define un objeto *RuleDSL* de nombre *dslCustomer000* con las siguientes expresiones válidas del lenguaje y su correspondiente traducción al lenguaje técnico de reglas:

Language Expression	**Mapping Expression**	**Scope**
Customer gender is "{gender}" and age is less than '{age}'	&c : Customer(CustomerGender == "{gender}" , CustomerAge <= {age})	CONDITION
Customer age is greater than '{age}'	&c : Customer(CustomerAge > {age})	CONDITION
Customer gender between '{startAge}' and '{finishAge}'	&c : Customer(CustomerAge >= {startAge} , CustomerAge <= {finishAge})	CONDITION
Customer gender between '{startAge}' and '{finishAge}'	&c : Customer(CustomerAge >= {startAge} , CustomerAge <= {finishAge})	CONDITION
Customer Type is '{type}'	&c : Customer(CustomerTypeid == {type})	CONDITION
– No Identifier	&cId:CustomerId < 1	CONDITION
– From Montevideo	CountryId == 1,CityId == 1	CONDITION
There is a Customer	&c : Customer( )	CONDITION
Log : "{message}"	Msg("{message}")	CONSEQUENCE
Set Customer Type to '{value}'	&c.CustomerTypeid = {value}	CONSEQUENCE
Set Customer Maximum Allowed to '{value}'	&c.CustomerMaxAllowed = {value}	CONSEQUENCE
Update the fact : '{variable}'	update(&{variable})	CONSEQUENCE
do not loop	no-loop true	KEYWORD
priority {v}	salience {v}	KEYWORD

El juego de reglas detallado anteriormente se puede reescribir de la siguiente manera:

```
// indico que se va a utilizar el lenguaje detallado anteriormente
expander dslCustomer000
```

```
rule "Todos los clientes de UY son B"
  do not loop
  priority 100
  when
    There is a Customer
    - From Montevideo
  then
    Log : "Todos los clientes de UY son B"
    Set Customer Type to '2'// B
end

rule "Clientes masculinos de hasta 20 años son de tipo A."
  do not loop
  when
    Customer gender is "M" and age is less than '20'
  then
    Log : "Clientes masculinos de edad menor a 20 son de tipo A."
    Set Customer Type to '1'// A
end

rule "Clientes femeninos de hasta 30 años son de tipo A."
  do not loop
  when
    Customer gender is "F" and age is less than '30'
  then
    Log : "Clientes femeninos de hasta 30 son de tipo A."
    Set Customer Type to '1'// A
  end

rule "Clientes de mas de 70 años son de tipo C."
  do not loop
  when
    Customer age is greater than '70'
  then
    Log : "Clientes de mas de 70 son de tipo C."
    Set Customer Type to '3'// C
end

rule "Clientes entre 50 y 60 años asignar el tipo C."
  do not loop
  when
    Customer gender between '50' and '60'
  then
    Log : "Clientes entre 50 y 60 asignar el tipo C."
    Set Customer Type to '3'// C
    Update the fact : 'c'
end

rule "Clientes de tipo C asignar 300 como máximo de compras."
  do not loop
  when
    Customer Type is '3'
```

```
  then
    Log : "Clientes de tipo C asignar 300 como máximo de compras."
    Set Customer Maximum Allowed to '300.00'
end
```

La ejecución es equivalente al caso detallado anteriormente dado que la definición del objeto *RuleDSL* simplemente permite especificar las reglas a un nivel de abstracción mayor. Internamente, cuando se salva el juego de reglas, se genera la traducción al lenguaje técnico GeneXus ya detallado. También se genera la representación del lenguaje de reglas en la plataforma de ejecución.

Consideraciones

Las reglas definidas se pueden analizar en forma centralizada en la tabla de decisión de la Figura 5.3.

	Cursos de Acción	Reglas					
		1	2	3	4	5	6
Condiciones	CustomerGender		Male	Female			
	CustomerAge		<=20	<=30	>70	between(50,60)	
	CustomerTypeId						C
Acciones	Asignar CustomerType B	X					
	Asignar CustomerType A		X	X			
	Asignar CustomerType C				X	X	
	Asignar máximo compras permitido						X

Figura 5.3.: Tabla de decisión del Escenario - Cliente

Notar que los cambios realizados no solo impactan a las reglas definidas inicialmente sino que también se realizan nuevas segmentaciones y restricciones que aplican sobre otros términos definidos en el dominio sin haber modificado una línea de código del programa que ejecuta.

5.2.3. Escenario 2 – Producto

El escenario relacionado con los productos tiene como objetivo inicializar el precio de un producto en base a diferentes restricciones relacionadas con la categoría y procedencia de los productos. Es muy similar al escenario anterior ("*Cliente*"); aunque en este caso se evalúan diferentes tipos de expresiones – en particular reglas de tipo *Producción* – para realizar los cálculos que detallan las reglas de negocio.

Además, se experimenta con un lenguaje de definición de reglas de "*alto nivel*", utilizando varias palabras clave en español para lograr una definición de regla de negocio entendible para un hispanohablante.

Al igual al escenario de uso del Cliente de la sección 5.2.2, se agregan y modifican reglas de negocio para satisfacer las nuevas restricciones que se agregan al caso de uso sin tener que modificar el código procedural del programa que ejecuta. En este caso es importante destacar la notoria mejora de legibilidad de las reglas de negocio entre lo que es el formato técnico versus la representación en "español" utilizando un lenguaje de alto nivel. Por más información consultar la sección G.2 del Anexo G .

5.2.4. Escenario 3 – Factura

El escenario relacionado con el registro de facturas tiene como objetivo aplicar diferentes tipos de *validaciones*, *descuentos* y *promociones* cuando se da de alta una factura, dejando abierta la puerta para variar dichas restricciones según sea necesario en tiempo de ejecución.

Resulta importante destacar que intervienen varios objetos en la evaluación de las reglas. Se tienen en cuenta varios elementos relacionados a la factura como es el cliente, las líneas de factura y los productos involucrados en dicha facturación y además se utilizan nuevos recursos del motor de evaluación y lenguaje de reglas, como ser `prioridad` para influir en el orden de ejecución de reglas, `agenda-group` para agrupar reglas que se deseen evaluar como un todo por bloques. Se suspende la ejecución del motor de reglas en la sección de consecuencias cuando se encuentra un error en los datos mediante la operación `Halt` y utiliza operadores lógicos como por ejemplo `not` y el cuantificador universal `exists`.

Notar que todas estas características se ocultan utilizando el lenguaje de reglas de alto nivel, manteniendo una legibilidad de las reglas de negocio acorde al experto del negocio. Por más información referirse a la sección G.2 del Anexo G.

5.2.5. Escenario 4 – Workflow

El escenario 4 explora el uso de reglas de negocio integrado a los procesos de negocio, de manera de lograr cierto control sobre lógica de negocio que influya en los flujos de procesos.

Supongamos un proceso de negocio como se detalla en la Figura 5.4 donde se generan pedidos que deben ser autorizados por un administrativo y es necesario toman algunas decisiones de cómo se sigue el flujo de proceso.

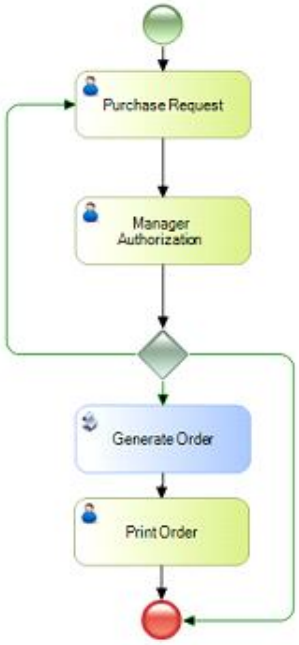


Figura 5.4.: Proceso de Negocio de ejemplo

En este caso en el condicional posterior a la tarea *Manager Authorization* interesa tener la capacidad de evaluar reglas de negocio para poder realizar fácilmente modificaciones en caso que sea necesario. Si bien el flujo de proceso es fijo, interesa poder controlar la lógica que gobierna los diferentes condicionales utilizando reglas de negocio. En el ejemplo, las reglas que rigen al condicional son las siguientes:

- *Regla 1*: Rechazar Orden de compra si la misma no está completa, por ejemplo tiene una descripción de pedido vacía.

```
rule "Rechazar"
  no-loop true
  salience 10
  when
    &myRequest:Request(ReqDsc=="")
  then
    Msg("Rechazar...")
    &myRequest.ReqAut = 2
    RuleEngine.Halt()
end
```

- *Regla 2*: Generar Orden de Compra si el total es mayor a 20.

```
rule "Generar Orden
  no-loop true
  when
    &myRequest:Request(&tot:ReqTotAmt > 20)
```

```
  then
    Msg("Generar Orden...")
    &myRequest.ReqAut = 1
    RuleEngine.Halt()
end
```

- *Regla 3*: Por defecto finalizar el flujo de ejecución.

```
rule "Finalizar"
  no-loop true
  salience -10
  when
    &myRequest:Request()
  then
    Msg("Finalizar...")
    &myRequest.ReqAut = 3
end
```

Evaluación

La evaluación del escenario se realiza en forma manual, ingresando inicialmente un requerimiento sin definir su descripción; se ejecuta la regla "Rechazar" al llegar al condicional correspondiente por lo que se ejecuta la decisión de volver al inicio del proceso a la tarea "*Purchase Request*".

Posteriormente se actualiza el requerimiento agregando varias líneas de manera de lograr que el total sea mayor a 20. En este caso, al evaluarse la decisión a través de la ejecución de reglas se ejecuta la regla "*Generar Orden*" siguiendo el flujo el proceso "*Generate Orden*" y posteriormente finaliza el flujo normal de ejecución de la instancia de proceso.

Notar que utilizando el recurso `salience` se define fácilmente el orden de precedencia de las reglas. La *Regla 1* es una validación por lo que tiene la prioridad mayor; la *Regla 2* representa la ejecución normal de la lógica del condicional y la *Regla 3* tiene la mínima prioridad y se utiliza como valor predeterminado cuando ninguna regla aplique.

Lenguaje de Alto nivel

Para las tres reglas detalladas anteriormente se desarrollan dos objetos *RuleDSL* para hacer pruebas con dos lenguajes diferentes, por un lado uno aproximado al lenguaje inglés y el segundo una aproximación al lenguaje japonés.

El objeto *dslFlowEnglish* que centraliza el lenguaje en "formato inglés" para manipular el proceso de negocio de ejemplo se detalla en la sección F.2 del Anexo F. Las reglas se reescriben de la siguiente forma:

```
expander dslFlowEnglish
rule "Reject"
 Do not loop
 when
  Request Description is Empty
 then
  Message "Reject..."
  Set Authorization Status "2"
end

rule "Generate Order"
 Do not loop
 when
  A Request
   - Total greater than 20
 then
  Message "Generate Order..."
  Set Authorization Status "1"
  Stop Execution
end

rule "GoToEnd"
 Do not loop
 Low Priority
 when
  A Request
 then
  Message "GoToEnd"
  Set Authorization Status "3"
end
```

Figura 5.5.: Reglas utilizando parametrización al inglés

La ejecución del caso de uso utilizando las nuevas expresiones del objeto *RuleDSL* es equivalente.

Al igual que con el escenario relacionado al Cliente (sección 5.2.2) en donde se experimenta con un lenguaje de reglas de alto nivel para hispanohablantes, en este caso se define además un nuevo objeto *dslFlowJapanese* donde se centraliza un lenguaje de reglas de ejemplo de alto nivel de abstracción para ser utilizado en el mercado japonés (sección F.2 del Anexo F):

```
expander dslFlowJapanese
rule "Reject"
 輪にならないでください
 when
  要請説明がむなしいです
 then
  メッセージ "Reject"
  セットされた委任ステータス "2"
end

rule "Generate Order"
 輪にならないでください
 when
  要請
   - より素晴らしいことを合計してください 20
 then
  メッセージ "Generate Order"
  セットされた委任ステータス "1"
end

rule "Ending"
 輪にならないでください
 when
  要請
 then
  メッセージ "Ending"
  セットされた委任ステータス "3"
  停止実行
end
```

Figura 5.6.: Reglas utilizando parametrización al japonés

La ejecución del escenario es equivalente utilizando cualquiera de los “dialectos” (lenguajes de alto nivel) detallados anteriormente al igual que con el lenguaje técnico detallado al inicio de la sección.

Consideraciones

Las decisiones del flujo de proceso se encuentran centralizadas en tres reglas de negocio, pudiéndose modificar según se requiera utilizando el editor de reglas (sección 4.8).

Se utiliza a las reglas de negocio como restricciones sobre los procesos empresariales, donde no es necesario conocer a priori el detalle de cómo va a funcionar el proceso, sino que se agrega un grado de libertad al centralizar ciertas decisiones y restricciones

a las reglas de negocio que podrán definirse en tiempo de implantación logrando una definición de procesos de negocio desestructurada.

Además, se experimenta la definición de dichas reglas en dos lenguajes de alto nivel de abstracción diferentes con el objetivo de simular la implantación de la aplicación en diferentes mercados. En este caso se utiliza la misma infraestructura en todas las implantaciones (ya que se trata de la misma aplicación) y en cada contexto sólo se debe personalizar en objetos de tipo *RuleDSL* el lenguaje que maneja el experto del negocio, lo que comúnmente se denomina *Application Localization*.

5.2.6. Escenario 5 – OAV

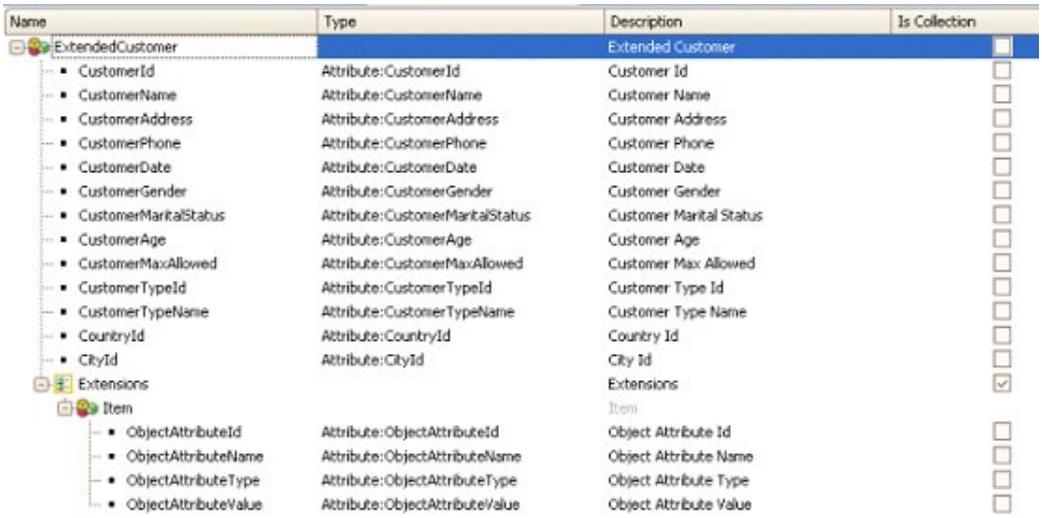
El modelo de datos OAV[2] (Objeto-Atributo-Valor) también conocido como EAV (Entidad–Atributo-Valor) permite extender la información de una base de datos sin necesidad de modificar la estructura de la misma ni realizar ningún tipo de reorganización física.

Se permite agregar atributos a una entidad de negocio en tiempo de ejecución; se amplía el modelo de datos de forma dinámica. Por ejemplo, en matemática a este tipo de modelo se le conoce como matriz dispersa.

A partir del caso de uso relacionado con el cliente (detallado en la sección 5.2.2), se experimenta con ésta forma de modelado y agrega una nueva entidad denominada *ExtendedCustomer* permitiendo agregar propiedades genéricas instanciables por instalación representadas en una colección de objetos.

Para diseñar una aplicación que se ajuste a estos requerimientos, diseñamos una entidad como detalla la Figura 5.7.

[2] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?1847`



Name	Type	Description	Is Collection
ExtendedCustomer		Extended Customer	☐
CustomerId	Attribute:CustomerId	Customer Id	☐
CustomerName	Attribute:CustomerName	Customer Name	☐
CustomerAddress	Attribute:CustomerAddress	Customer Address	☐
CustomerPhone	Attribute:CustomerPhone	Customer Phone	☐
CustomerDate	Attribute:CustomerDate	Customer Date	☐
CustomerGender	Attribute:CustomerGender	Customer Gender	☐
CustomerMaritalStatus	Attribute:CustomerMaritalStatus	Customer Marital Status	☐
CustomerAge	Attribute:CustomerAge	Customer Age	☐
CustomerMaxAllowed	Attribute:CustomerMaxAllowed	Customer Max Allowed	☐
CustomerTypeId	Attribute:CustomerTypeId	Customer Type Id	☐
CustomerTypeName	Attribute:CustomerTypeName	Customer Type Name	☐
CountryId	Attribute:CountryId	Country Id	☐
CityId	Attribute:CityId	City Id	☐
Extensions		Extensions	☑
Item		Item	
ObjectAttributeId	Attribute:ObjectAttributeId	Object Attribute Id	☐
ObjectAttributeName	Attribute:ObjectAttributeName	Object Attribute Name	☐
ObjectAttributeType	Attribute:ObjectAttributeType	Object Attribute Type	☐
ObjectAttributeValue	Attribute:ObjectAttributeValue	Object Attribute Value	☐

Figura 5.7.: Cliente Extendido

Notar que el cliente (le denominamos `ExtendedCustomer`) tiene almacenada la información básica de clientes (nombre, teléfono, género etc...) que será aquella información que es igual para todas las instalaciones de la aplicación y posteriormente define una lista de Items (propiedades genéricas representadas por la colección `Extensions`) que serán aquellas definiciones que podrán variar en las diferentes instalaciones de la aplicación.

Supongamos dos clientes de prueba (Adán y Eva) con tres atributos instanciados en tiempo de instalación:

- *Deporte preferido* utilizando el Identificador 1.
- *Altura* utilizando el Identificador 2.
- *Última actualización* utilizando el Identificador 3.

Según la estructura detallada anteriormente podría modelarse de la siguiente forma:

```
{
  // Adan
  CustomerId = 1
  CustomerName = "Adan"
  ......
  CityId = 1
  Extensions
  {
    Item.Insert(1,"1",AttributeType.Character,"Soccer")
    Item.Insert(1,"2",AttributeType.Numeric,"1.70")
    Item.Insert(1,"3",AttributeType.Date,"01/03/2000")
  }
```

```
}

{
  // Eva
  CustomerId = &CustomerId
  CustomerName = &CustomerName
  ......
  CityId = 1
  Extensions
  {
    Item.Insert(1,"1",AttributeType.Character,"Gym")
    Item.Insert(1,"2",AttributeType.Numeric,"1.63")
    Item.Insert(1,"3",AttributeType.Date,"12/03/2000")
  }
}

Subgroup Item( &ObjectAttributeId, &ObjectAttributeName,
               &ObjectAttributeType, &ObjectAttributeValue )
  Item
  {
    ObjectAttributeId = &ObjectAttributeId
    ObjectAttributeName = &ObjectAttributeName
    ObjectAttributeType = &ObjectAttributeType
    ObjectAttributeValue = &ObjectAttributeValue
  }
Endsubgroup
```

Utilizando el esquema de definición de reglas de negocio se desea tener la capacidad de definir reglas sobre los atributos dinámicos. Esto quiere decir que es necesario manipular la lista de `Extensiones` del objeto `ExtendedCustomer` para poder definir dichas restricciones sobre los "*atributos dinámicos*" que se definan.

Supongamos que se quiere aplicar una regla para el cliente de nombre "*Adan*" y que tenga definido el atributo dinámico "*Deporte preferido*", la especificación de la regla podría definirse como:

```
rule "Adan Found"
  when
    &c:ExtendedCustomer(&name:CustomerName=="Adan",&l:Extensions)
    &i:ExtendedCustomer.Item(ObjectAttributeName=="1") from &l
  then
    Msg("Hola Adan:"+&c.ToXML())
end
```

En la sección de condiciones, la primera restricción simplemente filtra por el nombre del cliente y proyecta en `&l` la colección de extensiones. La segunda restricción se encarga de filtrar por el identificador "1" dentro de la colección para obtener la representación interna de "`Deporte preferido`", es decir busca un objeto con ese identificador dentro de la colección de extensiones.

Se detectó que este tipo de regla no es posible definirla debido a como se manejan las colecciones de elementos desde GeneXus a nivel del código generado en la plataforma y cómo funciona el "matcheo" de patrones en el motor de reglas utilizado.

Por un lado la generación de código GeneXus al manipular colecciones de elementos la realiza con objetos genérico. Esto quiere decir que para referenciar un elemento dentro de la colección es necesario realizar un *Cast*[3] al tipo de objeto destino para poder trabajar con las propiedades y métodos de los elementos. En el contexto de las reglas dinámicas, donde se desea definir restricciones sobre atributos dinámicos implica conocer el tipo del objeto que fue almacenado en la colección para poder resolver la correspondiente conversión y realizar el filtro deseado.

Es decir, supongamos el siguiente código GeneXus que define un atributo dinámico y lo agrega a una colección:

```
&Extension01 = new()
&Extension01.ObjectAttributeId = 1
&Extension01.ObjectAttributeName = "1"
&Extension01.ObjectAttributeType = AttributeType.Character
&Extension01.ObjectAttributeValue = "Soccer"
&ExtendedCustomer.Extensions.Add(&Extension01)
&Extension01 = &ExtendedCustomer.Extensions.Item(1)
```

siendo `&ExtendedCustomer` de tipo `ExtendedCustomer` y `&Extensions01` de tipo `ExtendedCustomer.Item`. La generación de código para la plataforma Java es la siguiente:

```
AV65Extension01 = (SdtExtendedCustomer_Item)new SdtExtendedCustomer_Item(
remoteHandle, context);
AV65Extension01.setgxTv_SdtExtendedCustomer_Item_Objectattributeid(1);
AV65Extension01.setgxTv_SdtExtendedCustomer_Item_Objectattributename("1");
AV65Extension01.setgxTv_SdtExtendedCustomer_Item_Objectattributetype(0);
AV65Extension01.setgxTv_SdtExtendedCustomer_Item_Objectattributevalue("Soccer");
AV64ExtendedCustomer.getgxTv_SdtExtendedCustomer_Extensions().add( AV65Extension01,
0);
AV65Extension01 = (SdtExtendedCustomer_Item)
 (
  (SdtExtendedCustomer_Item)
   AV64ExtendedCustomer.getgxTv_SdtExtendedCustomer_Extensions().elementAt(-1+1)
 );
```

Notar que para poder referenciar un elemento de tipo `ExtendedCustomer.Item` es necesario realizar un *cast* una vez que se obtiene el elemento de la colección correspondiente (detallado en la última línea de la sección de código anterior); que se corresponde con el código GeneXus:

```
&Extension01 = &ExtendedCustomer.Extensions.Item(1)
```

[3] http://en.wikipedia.org/wiki/Type_conversion

La herramienta utilizada (Drools 5.0) no permite definir casts explícitos en las condiciones[4] de las reglas por lo que no se permite condicionar reglas que involucren estos atributos dinámicos.

Para solucionar el problema en forma temporal y validar el escenario de uso se crea un objeto "*Helper*" GeneXus denominado `HasExtendedProperty`, que resuelve con código GeneXus lo que no puede resolverse desde la plataforma.

El programa `HasExtendedProperty` recibe como argumento un objeto de tipo `ExtendedCustomer` y un nombre de propiedad dinámica retornando un tipo de dato *Boolean* (verdadero o falso) indicando si la propiedad existe en la colección de "atributos dinámicos" del cliente. Por lo que para definir una condición de una regla sobre un atributo dinámico se utiliza un procedimiento GeneXus como si fuera una función.

La firma del objeto "`HasExtendedProperty`" es la siguiente:

```
Parm(in:&ExtendedCustomer,in:&ObjectAttributeName,out:&result);
```

y ejecuta el siguiente código GeneXus:

```
&result = false
For &Property in &ExtendedCustomer.Extensions
  If (&Property.ObjectAttributeName = &ObjectAttributeName)
    &result = true
    return
  endif
endfor
```

La regla de negocio se puede definir de la siguiente manera, utilizando el nuevo objeto `HasExtendedProperty`; se tiene que utilizar la función `eval` (sección 4.3.3) para procesar el resultado de la evaluación del procedimiento en la sección de condiciones, es decir:

```
rule "Adan Found"
  when
    &c:ExtendedCustomer(&name:CustomerName=="Adan",&l:Extensions)
    eval(HasExtendedProperty(&c,"1"))
  then
    Msg("Hola Adan:"+&c.ToXML())
End
```

Debido a restricciones a nivel de la gramática en el lenguaje de reglas GeneXus definido (ver Anexo F) no es posible salvar dicha regla debido a que la función `eval` no fue implementada en su totalidad. La restricción no permite que la función `eval`

[4] `http://osdir.com/ml/java.drools.user/2007-01/msg00204.html`

reciba cualquier tipo de expresión, como es el caso de una llamada a un objeto con varios argumentos retornando un valor booleano.

Igualmente para validar el escenario se realizan pruebas sobre el lenguaje de definición de reglas *DRL* directamente en la plataforma de ejecución, pudiéndose definir la regla de la siguiente forma:

```
rule "Adan Found"
  when
    $c:com.sample.SdtExtendedCustomer(
        $name:gxTv_SdtExtendedCustomer_Customername=="Adan",
        $l:gxTv_SdtExtendedCustomer_Extensions )
    eval(new com.sample.hasextendedproperty(-1).executeUdp($c,"1"))
  then
    com.ruleset.api.MessageHelper.msg(
      "Hola Adan:"+
      $c.toxml(false,"ExtendedCustomer","brsamplexev101")
    );
end
```

Utilizando el juego de datos detallado al inicio de la sección se ejecuta correctamente la regla sobre el *cliente extendido* "Adan" con el atributo dinámico `Deporte preferido`.

```
Hola Adan:<ExtendedCustomer xmlns="brsamplexev101">
  <CustomerId>1</CustomerId>
  <CustomerName>Adan</CustomerName>
  <CustomerAddress>Adan</CustomerAddress>
  <CustomerPhone>Adan</CustomerPhone>
  ........................
  <Extensions>
    <Item>
      <ObjectAttributeId>1</ObjectAttributeId>
      <ObjectAttributeName>1</ObjectAttributeName>
      <ObjectAttributeType>0</ObjectAttributeType>
      <ObjectAttributeValue>Soccer</ObjectAttributeValue>
    </Item>
    <Item>
      <ObjectAttributeId>1</ObjectAttributeId>
      <ObjectAttributeName>2</ObjectAttributeName>
      <ObjectAttributeType>1</ObjectAttributeType>
      <ObjectAttributeValue>1.70</ObjectAttributeValue>
    </Item>
    <Item>
      <ObjectAttributeId>1</ObjectAttributeId>
      <ObjectAttributeName>3</ObjectAttributeName>
      <ObjectAttributeType>2</ObjectAttributeType>
      <ObjectAttributeValue>01/03/2000</ObjectAttributeValue>
    </Item>
  </Extensions>
</ExtendedCustomer>
```

Consideraciones

El caso de uso OAV no se pudo completar en su totalidad utilizando código GeneXus. Inicialmente se detectó una restricción en la forma de manipular las colecciones de objetos en el motor de reglas utilizado en relación a la generación de código GeneXus para la plataforma Java. Se decide utilizar código GeneXus para poder solucionar el caso y poder definir condiciones de una regla que involucren colecciones de objetos.

Se encuentra una nueva restricción, en este caso en la definición de la gramática del lenguaje de reglas diseñado ("GXRL") donde no se soporta la evaluación de condiciones complejas a partir de las producciones que dependan de la función *eval*.

Sin embargo, manipulando directamente el lenguaje de reglas y código generado GeneXus se pudo ejecutar la regla de negocio. Por lo tanto, es viable resolver casos de uso de éste tipo realizando modificaciones a nivel de código generado o agregando nuevos artefactos GeneXus para poder solventar el problema desde el lenguaje de reglas diseñado. Se utilizó dicho enfoque para finalizar el escenario agregando "Helper Objects[5]" en GeneXus, los cuales se utilizan desde las condiciones de reglas con el objetivo de manipular colecciones de objetos.

5.2.7. Escenario 6 – Interacción con código GeneXus

Un recurso muy útil que es importante en el contexto de ejecución de reglas, es tener la posibilidad de ejecutar procedimientos GeneXus para reutilizar lógica de negocio ya existente.

Este escenario plantea la investigación de cómo lograr ejecución de objetos GeneXus desde la consecuencia de reglas. Para ello se reutiliza parte del caso de uso de facturación (sección 5.2.4), especificando el siguiente juego de reglas a utilizarse con el mismo juego de datos.

```
rule "Invoice02 - Call procedure"
  when
  &i8 : Invoice(
      &CustomerId:CustomerId==1 ,
      &InvoiceAmount:InvoiceSubTotal ,
      &InvoiceDate:InvoiceDate ,
      &CustomerName:CustomerName)
  then
    Msg("Invoice02 - Call procedure")
    if (1==1)
      // Modifico el cliente predeterminado
      &i8.CustomerId = 2
    else
      &i8.CustomerId = 1
```

[5] http://forums.asp.net/t/488434.aspx

```
    endif
    // Creo una linea de factura
    &itemInvoiceLine8 = new Invoice.Line()
    &itemInvoiceLine8.InvoiceLineId = 100
    &itemInvoiceLine8.ProductId = 6
    &itemInvoiceLine8.InvoiceLineQty = 1
    &itemInvoiceLine8.InvoiceLineProductPrice= 1.00
    &itemInvoiceLine8.InvoiceLineTaxes = 0.01
    &itemInvoiceLine8.InvoiceLineDiscount = 0.01
    &itemInvoiceLine8.InvoiceLineAmount = 1.00
    // Llamo a un objeto GX
    gxProcedure01.Call(&i8, &itemInvoiceLine8, &CustomerId, &InvoiceAmount
,&CustomerName, true)
    gxLog.Call(&i8.InvoiceId,"Invoice02 - Call procedure")
end
```

La sección de acciones de la regla entre otras cosas utiliza la construcción `If...then...else`, creación de objetos de negocio como es el caso del constructor "`new Invoice.Line()`", asignación de propiedades y finalmente ejecución de procedimientos GeneXus como son los objetos `gxLog` y `gxProcedure01`.

Evaluación

Reutilizando el juego de datos del escenario de facturación se obtiene el siguiente resultado:

```
Invoice02 - Call procedure
gxProcedure01 is being executed.
Insert OK
<Invoice xmlns="brsamplexev101">
  <InvoiceId>620</InvoiceId>
  <InvoiceDate>2009-10-08</InvoiceDate>
  <CustomerId>2</CustomerId>
  <CustomerName>Customer 0</CustomerName>
  <Line>
    <Invoice.Line xmlns="brsamplexev101">
      <InvoiceLineId>1</InvoiceLineId>
      ..................
```

Interesa resaltar que el mensaje "`gxProcedure01 is being executed`" se llama desde el código del objeto `gxProcedure01`, que contiene el siguiente código GeneXus:

```
&msg = &Pgmname.Trim() + " is being executed."
msg(&msg,status)
```

El objeto `gxLog` también fue ejecutado ya que se insertó un registro en una tabla de log genérica tal como detalla el código procedural del objeto correspondiente:

```
&LogDsc = Format("%1:%2",&id,&message)
&date = servernow()
// Log Message
New
  LogDsc = &LogDsc
  LogDate = &date
  When Duplicate
  // Do nothing, is an autonumber
EndNew
```

Listando la tabla de log se evidencia el siguiente registro resultado de la ejecución:

ID	Description	Timestamp
28	625:Invoice02 – Call procedure	01/12/2009 09:00 AM

La columna *ID* es un atributo autonumber, la columna *Description* se corresponde con el atributo *LogDsc*, y la columna *Timestamp* se corresponde con el atributo *LogDate*.

Consideraciones

Se logra ejecutar en forma transparente objetos GeneXus de tipo procedimiento; solo se implementa el pasaje de parámetros de tipo *In*. Queda fuera del escenario realizar pruebas de ejecución sobre objetos de tipo DataProvider, ya que en este caso el pasaje de parámetros es *InOut* en forma predeterminada (detallado en la sección 4.12.1).

5.2.8. Escenario 7 – Modelo Universal de Datos

Los modelos universales de datos (UDM - Universal Data Model) son modelos de datos que apuntan a almacenar cualquier tipo de información sin necesidad de modificar el esquema de la base de datos.

Todos estos modelos de datos tienen un enfoque común respecto a como descomponen la información, generalmente en:

- *Entidades*: representan objetos del mundo real, como por ejemplo: Cliente, Persona, Factura, Producto.
- *Atributos*: Las entidades tienen "atributos". En general dos objetos de la misma *Entidad* tendrán atributos más o menos parecidos, el valor de los atributos es lo que diferencia un objeto de otro.
- *Relación entre Entidades*: Cada *Entidad* no está aislada de su contexto; se relaciona de diferente forma con otras entidades. Una *Factura* está relacionada con el *Cliente* y *Producto*, a su vez el *Producto* está relacionado con *Categoría de Producto* y así sucesivamente.

El escenario OAV resuelve el problema de extender las propiedades de cualquier entidad de un dominio, sin embargo no resuelve escenarios en donde se desea agregar y relacionar nuevas entidades; es decir poder instanciar "Entidades" que no fueron previstas inicialmente en el sistema; pudiendo relacionarlas con entidades ya existentes. A este tipo de modelado flexible se le denomina modelo universal de datos (UDM - Universal Data Model).

Se experimenta definiendo reglas de negocio en formato técnico sobre una entidad de ejemplo denominada *EntitySDT* que representa a cualquier tipo de entidad según detalla la Figura 5.8.

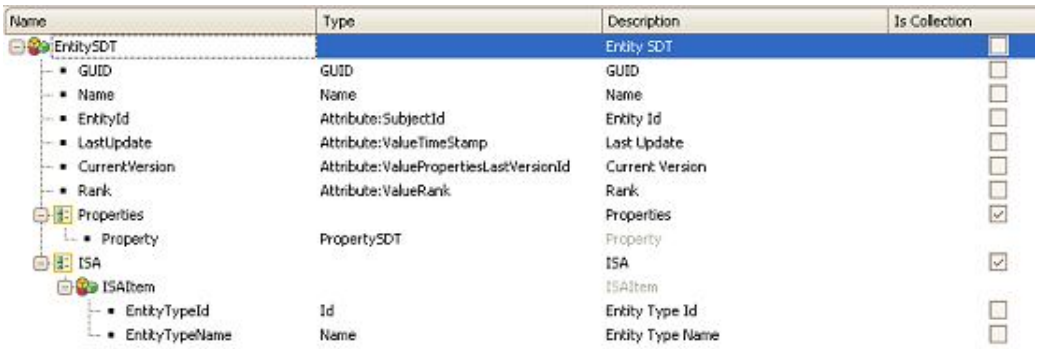


Figura 5.8.: Entidad genérica EntitySDT

Define en términos generales información básica, un puntero a otros objetos que describen la entidad (atributo *EntityID*) y una lista de propiedades complejas (propiedades compuestas por otras propiedades) representada por las colecciones *Properties* e *ISA*.

Al igual que el escenario OAV (sección 5.2.6), el caso de uso no se pudo completar en su totalidad con código GeneXus dado que "sufre" de los mismos problemas encontrados en el escenario OAV; en el sentido que para poder referenciar tanto propiedades como relaciones, en las condiciones de las reglas se tiene que manipular colecciones de objetos.

Solventando las limitaciones definidas en la gramática y utilizando "*Helper Objects*" para utilizar colecciones en condiciones de reglas, es viable especificar cualquier tipo de regla sobre entidades genéricas definidas en un modelo universal de datos.

5.2.9. Escenario 8 – Uso de Operaciones

Existen varios métodos que se pueden utilizar por código desde GeneXus para poder informar al motor de evaluación de reglas del cambio de estado de diferentes objetos. Basados en el caso de uso de la sección 5.2.1 (*Hola Mundo*), se agregan métodos al objeto externo *RulesetEngine* para utilizar las operaciones `Update` y `Retract`.

La ejecución del escenario fue satisfactoria. Interesa resaltar la modificación del objeto `MyMessage` desde las consecuencias de reglas y código GeneXus informando al motor de reglas de la actualización del objeto con la operación `Update` lográndose que se ejecuten nuevamente las reglas.

Al final de la ejecución de la prueba se utiliza la operación `Retract` y al ejecutar las reglas se dispara la regla "`No hay Mensaje`" indicando que ya no existen objetos de tipo `MyMessage` en la memoria de trabajo del motor, referirse al Anexo G por más información.

5.2.10. Escenario 9 – "Truth Maintenance"

Al motor de evaluación de reglas también se lo puede utilizar como mecanismo de inferencia o razonamiento. El mecanismo *Truth Maintenance* [142, 207] es un recurso que proveen los motores para eliminar hechos de la memoria de trabajo en forma automática cuando ciertas hipótesis ya no son válidas, generalmente implementadas mediante la operación *InsertLogical*.

Dicho operador es similar a la operación `insert` teniendo en cuenta que el objeto insertado será eliminado en forma automática de la memoria de trabajo cuando no existan más hechos que soporten la "verdad" de la regla que hizo la inserción lógica. En todo momento se cumplen las siguientes condiciones:

- El objeto es único.
- La validez del objeto es mantenida en forma automática.

Truth Maintenance es un recurso interesante para especificar dependencias lógicas entre hechos. Todos los hechos que se insertan en la sección derecha de una regla (acciones) utilizando el operador *InsertLogical*, se tornan dependientes de la sección izquierda (condiciones). En caso que alguna de las condiciones se tornen inválidas, los hechos dependientes se eliminan en forma automática.

Supongamos el siguiente ejemplo que se basa sobre el escenario de uso básico *Hola Mundo* (sección 5.2.1). Se parte de los siguientes mensajes que son insertados en la memoria de trabajo del motor:

```
MyMessages
{
  MyMessage
  {
    Message = "1. Mensaje OK"
    Status = true
  }
  MyMessage
  {
    Message = "2. Mensaje FALSE"
```

```
    Status = false
  }
  MyMessage
  {
    Message = "3. Mensaje FALSE"
    Status = false
  }
}
```

A partir de tres objetos de tipo mensaje, el objetivo es lograr que todos los mensajes pasen de status `false` a `true`; en este caso los últimos dos mensajes.
Para ello se utiliza el recurso inserción lógica, cuando existe un mensaje falso se inserta un objeto de tipo *Alerta*.

```
rule "Mensaje Incorrecto"
  salience 10
  when
    exists( &m:MyMessage( Status == false ) )
  then
    &vAlert = new Alert()
    insertLogical( &vAlert )
end
```

Se crea una regla que muestra un mensaje indicando que se tienen que procesar las alertas:

```
rule "Alertas"
  salience 10
  when
    exists( Alert() )
  then
    Msg("Hay Alertas a procesar")
end
```

Se crea otra regla para "arreglar los problemas", en este caso se actualiza el status de los mensajes, notar que en las condiciones se está apoyando en el objeto *Alerta* insertado en forma lógica utilizando el cuantificador existencial `exists` (detallado en la sección 4.3.3):

```
rule "Arreglando Problemas..."
  when
    &m: MyMessage( Status == false )
    exists( Alert() )
  then
    Msg( "Arreglando " + &m.Message)
    &m.Status = true
    update(&m)
end
```

Se crea una regla que aplica cuando ya no existen alertas para procesar:

```
rule "Ya no hay Alertas"
  when
    not( Alert() )
  then
    Msg( "No hay Alertas!!!")
end
```

Es en la regla "`Mensaje Incorrecto`" en donde se declara la dependencia lógica de las condiciones. Mientras exista al menos un mensaje (objeto `MyMensaje`) con status `false` se tiene que mantener la alerta.

Con la regla "`Arreglando problemas`" se emula la situación en donde se modifican las propiedades de los mensajes para dejarlos en un status válido. Al procesar los dos mensajes con status `false` ya no se puede soportar la inserción lógica por lo que en forma automática se hace `retract` del objeto *Alerta*; esto trae como consecuencia que se disparen nuevas reglas, en este caso, "`Ya no hay Alertas`".

El resultado en ejecución es el siguiente:

```
\========== Execution started ==========
"C:\Archivos de programa\Java\jdk1.6.0\bin\java.exe" com.sample.atms02

inserting 1. Mensaje OK
inserting 2. Mensaje FALSE
inserting 3. Mensaje FALSE
Hay Alertas a procesar
Arreglando 3. Mensaje FALSE
Arreglando 2. Mensaje FALSE
No hay Alertas!!!
Execution Success
```

Consideraciones

Este recurso es muy utilizado por ejemplo para detectar transacciones con probabilidad de fraude. Se especifican reglas que ante cierta casuística simplemente hacen inserciones lógicas de otros objetos. Posteriormente, otras reglas que trabajan sobre los objetos originales y lógicos deberán validar dichas sospechas. Es el propio motor que mantiene la validez de las "alertas" según las modificaciones que se hagan en la memoria de trabajo.

En el ejemplo planteado se verifican tres bloques de ejecución claros de disparo de reglas utilizando este recurso:

- Ejecución de reglas que realizan la inserción lógica de otros objetos.

- Activación y ejecución de reglas que realizan operaciones mientras la condición lógica sea válida.
- Activación y ejecución de reglas cuando se vuelve a un estado “normal” de ejecución.

6. Conclusiones

El trabajo realizado consta de tres partes fundamentales, inicialmente comprender la propuesta de administración de reglas de negocio aplicada al desarrollo de software con el objetivo de lograr aplicaciones flexibles, fácilmente modificables y adaptables al cambio. Posteriormente, realizar una propuesta que utilice dicho enfoque en lo que es la construcción de Sistemas de Información en el ecosistema GeneXus. Finalmente, realizar una evaluación por medio de un prototipo y juego de casos de uso del aporte de la propuesta.

6.1. Estado del Arte

Al comenzar el proyecto se estudia el enfoque de administración de reglas de negocio tanto desde un punto de vista académico como de la industria. Se busca comprender la motivación por la cual surge el enfoque, cuáles son los problemas que intenta resolver, y cómo propone hacerlo. En términos generales, podemos decir que se busca la construcción de aplicaciones con alta adaptación y flexibilidad.

Para ello se posiciona a las reglas de negocio como una característica esencial, una cualidad fundamental del software que debe de expresarse en forma explícita e independiente de la funcionalidad principal de cualquier Sistema de Información. Se plantea que es primordial realizar esta separación para reflejar el hecho que las reglas de negocio no necesariamente evolucionan de la misma forma, a la misma velocidad que las funcionalidades del sistema.

El objetivo es separar cierta parte de la lógica de negocio; aquellos puntos de decisión que sean convenientes que queden definidos externos al software en sí, especificados como reglas de negocio. De esta manera, la adaptación y agilidad frente a cambios que se deban realizar, se logra modificando las reglas de negocio administradas en forma externa a la aplicación.

A su vez, con este mecanismo no solo se formaliza, centraliza y garantiza el cumplimiento de políticas y restricciones empresariales clave, también involucra a otros actores de la organización, fundamentalmente los expertos del negocio, brindando capacidades para que puedan entender e influir en los sistemas a través de la manipulación de reglas de negocio.

6.2. Propuesta

Antes de presentar la propuesta en sí, se realiza una breve reseña de la herramienta GeneXus. Para el entendimiento de cómo funciona la plataforma GeneXus se hace una comparación con el enfoque de desarrollo de software basado en modelos (Model Driven Engineering) [170], una metodología "equivalente" al enfoque GeneXus [178] en donde la realidad se describe en una Base de Conocimiento.

A partir del conocimiento del enfoque BRA (Business Rules Approach) [204] y GeneXus [178] se realiza una propuesta que use dicho enfoque (administración de reglas dinámicas) para utilizarse en el ecosistema GeneXus con el objetivo de lograr mayores niveles de flexibilidad y adaptabilidad de las aplicaciones generadas.

Para lograr una independencia real entre reglas y programas, se desarrolla un lenguaje de definición de reglas de negocio independiente de la tecnología, tomando como referencia para su construcción: los lenguajes específicos de dominio (Domain Specific Languages) [169]. También se analiza la extensa bibliografía relacionada con las diferentes taxonomías y categorías de reglas, estándares existentes, así como también lenguajes de reglas utilizados por las principales herramientas del mercado.

Se diseña un lenguaje de reglas genérico que se divide en dos partes. Por un lado, un lenguaje técnico de reglas a nivel del desarrollador GeneXus que representa el formato ejecutable del lenguaje de reglas de negocio, y por otro lado, un lenguaje de mayor nivel de abstracción personalizable que utiliza el lenguaje anterior como plataforma pero que tiene el objetivo de lograr un acercamiento con los expertos de negocio; lograr definir un lenguaje común (lenguaje natural si es posible) con el cual comunicar las reglas del negocio.

Para el lenguaje técnico de reglas de negocio, se toma como referencia las *reglas de producción* representadas por el estándar *PRR* (Production Rule Repesentation) [71] y el lenguaje de definición de regla DRL (Drools Rule Language) [13]. La segunda dimensión de la representación de reglas de "alto nivel" de abstracción, se enfoca en facilitar la tarea de entendimiento y definición de reglas de negocio por el experto del dominio. Para este lenguaje de "alto nivel" se toman ideas fundamentalmente del estándar SBVR (Semantics of Business Vocabulary and Business Rules) [93] impulsado por la OMG [57], el concepto de vocabulario y BOM (Business Object Model) de la metodología ABRDM [63].

El vocabulario es la verbalización lo más cercana posible al lenguaje natural de los conceptos y datos del modelo de negocio, y es lo que permite a los expertos del dominio expresar sus propias reglas de negocio. Al elevar el nivel de abstracción del lenguaje de reglas se facilita el entendimiento por parte del experto, pudiendo expresar las reglas en los términos del propio dominio. Esto permite que el experto utilice un lenguaje controlado donde puede agregar, modificar o eliminar reglas, influyendo en forma directa en la aplicación.

6.3. Evaluación

Se define un juego de escenarios basados en un ejemplo de facturación de una empresa tipo, en donde es interesante aplicar el enfoque de reglas de negocio; es decir, lograr modificar comportamiento una vez que la aplicación está en funcionamiento.

Se desarrolla un prototipo en donde se implementa la propuesta inicial y evalúa según los diferentes casos de uso.

Para proveer la interfaz "alto nivel" de reglas, se crea el objeto *RuleDSL* que satisface la definición de BOM (Business Object Model) y permite definir las reglas en un pseudo-lenguaje natural totalmente parametrizable y extensible. Dicha representación tiene una traducción directa a un lenguaje de reglas técnico que se refleja en un nuevo objeto GeneXus denominado *Ruleset*.

Se logra que las reglas de negocio se definan en forma ortogonal a los programas, quedando desacopladas de la tecnología que las implementa. Esta separación e independencia hacen que se tenga una mayor visibilidad para todos los expertos del negocio en lo que se refiere a la administración de restricciones clave, devolviéndose el "*control*" a los expertos del negocio. De esta manera, se logra un mecanismo de comunicación con el personal técnico.

Una vez declaradas las reglas, el próximo paso consiste en lograr la evaluación de las mismas por parte de la aplicación en los puntos de ejecución definidos. Para ello se utilizan motores de evaluación de reglas (BRE – Business Rules Engine) [135] ya existentes.

El motor de evaluación de reglas es el componente que ejecuta las reglas a partir de la especificación de alto nivel realizada por parte del experto. En el contexto del prototipo, se desarrollan mecanismos de traducción de reglas de negocio a partir del lenguaje propuesto para la plataforma GeneXus hasta llegar al lenguaje de la plataforma de ejecución, investigando cómo lograr una buena integración a nivel de código generado.

Para validar el funcionamiento del prototipo se implementan los diferentes casos de uso, detallando cómo se logra la flexibilidad, agilidad y adaptación al cambio según los requerimientos planteados.

6.4. Resultados alcanzados

Se alcanzaron los objetivos propuestos, lográndose un prototipo en donde las definiciones de reglas ocupan un lugar principal en el modelado de la aplicación expresando lo que realmente es importante a un nivel de abstracción acorde, ocultando detalles innecesarios.

El objetivo inicial consiste en lograr un prototipo funcional donde se permita experimentar con los escenarios propuestos en los tres lenguajes de generación

principales de la plataforma GeneXus: C#, Java, Ruby. En este contexto, se experimenta con tecnología existente en base a los primeros casos de uso propuestos utilizando los generadores GeneXus para la plataforma Web: Java, C# y Ruby seleccionando los motores de reglas Drools [13], Drools.Net [14] y Ruleby [85], respectivamente. El resultado de la evaluación de las herramientas es satisfactorio a nivel de código generado.

Se comienza la integración del prototipo al ecosistema GeneXus por la plataforma Java debido a que Drools [13] es la herramienta que mejores resultados obtuvo en la experimentación con código generado.

Al finalizar la fase de prototipación en Java logrando que los ejemplos básicos ejecuten de forma totalmente integrada al entorno de desarrollo GeneXus, se decide priorizar el desarrollo de más casos de uso para lograr una visión más clara del potencial de la tecnología. Es por ello que no se implementa la integración al prototipo de los ambientes de ejecución C# y Ruby según las investigaciones realizadas.

Cabe destacar que si bien existe una importante cantidad de herramientas en el mercado utilizando el enfoque BRA [204], no existe ningún tipo de estandarización a nivel de lenguaje de reglas de negocio. En este sentido se piensa que en este trabajo se está realizando un aporte al tema, dada la investigación realizada sobre los lenguajes de definición de reglas de negocio, tanto a nivel comercial como estándares, desarrollando una propuesta independiente de la plataforma de ejecución que integra conceptos de ambos mundos implementada para la plataforma GeneXus.

Desde el punto de vista de la tecnología que sustenta a los motores de evaluación de reglas (BRE - Business Rule Engine) [135], existe un denominador común que queda demarcado principalmente por el uso del algoritmo Rete [150] y sus variaciones. La tecnología no es nueva y los componentes fundamentales que sustentan el enfoque no han cambiado desde hace algún tiempo. Ocasionalmente, aparecen mejoras o nuevos algoritmos relacionados con el procesamiento de reglas, pero en general el núcleo de ésta tecnología es estable. Sin embargo, cada producto además implementa diferentes funcionalidades, por lo que esto hace más difícil satisfacer la característica de *Universalidad* GeneXus en el sentido que, ofrecer una característica de evaluación de reglas dinámicas en el ecosistema GeneXus común para los diferentes lenguajes de generación (C#, Java y Ruby) implica lograr exponer el denominador común de la funcionalidad a nivel de abstracción GeneXus. Posteriormente, implica desarrollar para cada plataforma la correspondiente solución utilizando las herramientas seleccionadas.

6.5. Problemas Encontrados

Se detalla a continuación un resumen de los problemas encontrados a lo largo del proyecto.

6.5.1. Gramática no Extensible

Es importante destacar que actualmente no existe interoperabilidad entre diferentes motores de reglas, ya que no existe ningún estándar que se esté utilizando. Si bien existen varios intentos de estandarización sobre todo en lo que tiene que ver con los lenguajes de definición de reglas, la mayoría de los productos utilizan sus propios lenguajes propietarios.

Para el diseño del lenguaje de reglas para la plataforma GeneXus, se debió evaluar varios estándares, tomando ideas de los diferentes enfoques. No se pudo reutilizar la gramática GeneXus debido a que no es extensible; era deseable poder reutilizar las producciones relacionadas con la ejecución de código procedural para utilizarse en la sección de acciones de la definición de reglas de negocio.

Se definió una gramática independiente para modelar el concepto de reglas. Sin embargo, se mantuvo una sintaxis similar para mantener una sensación de integración con los demás objetos GeneXus dentro del ambiente de desarrollo, también disponible en el entorno de ejecución a través del editor externo de reglas de negocio.

6.5.2. Generador no Reusable

Una vez resuelto el lenguaje de reglas, se investigó el proceso de generación de artefactos para la plataforma – en este caso para el lenguaje de generación Java. Este proceso se dividió en dos puntos: por un lado se desarrolló un traductor que, a partir de la representación GeneXus de una regla, genera la regla en lenguaje de la plataforma, en este caso Drools 5.0. El segundo punto implica traducir los objetos GeneXus a la representación en la plataforma, de manera que las reglas puedan referenciar los objetos generados.

La reutilización de rutinas de especificación y generación para lograr generar las reglas de negocio fue descartada debido a que sería necesario realizar cambios sustanciales a nivel de especificación y generación (detallado en la sección 4.9.2). Se toma un camino alternativo en la generación del código que representa a las reglas en la plataforma. Se genera una metadata a partir del conocimiento GeneXus (sección 4.7) con todos los objetos del dominio relevantes para las reglas de negocio y se exportan como XML [18]. Utilizado un árbol sintáctico se traducen las reglas al lenguaje de la plataforma destino.

Este mecanismo, si bien es funcional, tiene la debilidad de que cualquier cambio en los criterios de generación de los objetos del negocio impactará de manera directa en la generación de los objetos relacionados con las reglas de negocio, teniendo que reflejar dichos cambios en la traducción de reglas en el ambiente de desarrollo GeneXus y en el editor externo de reglas.

6.5.3. Traducción a la Plataforma de ejecución

Se detectaron obstáculos importantes en la traducción semántica a nivel de abstracción GeneXus hacia la plataforma Java; un ejemplo claro es el uso y traducción de expresiones. Supongamos que se tiene la siguiente expresión en GeneXus que representa una acción para una regla de producción:

```
&p.Productprice = &p.Productprice + &p.Productbase * 0.01
```

en donde `p` es un objeto que representa un Producto, `Productprice` y `Productbase` representan dos de sus propiedades.

La traducción a la plataforma debe utilizar setter y getters para acceder a las propiedades del objeto "`p`" (especificación JavaBeans [38] para referenciar las propiedades de un objeto); es por ello que el primer punto que se debe resolver es el pasaje de propiedades a métodos teniendo en cuenta asignaciones, expresiones compuestas y balanceo de paréntesis.

También es necesario conocer los mapeos de los tipos de datos en la plataforma destino; esto quiere decir, a partir de la representación GeneXus de las propiedades de `&p`:

	ProductBase	**ProductPrice**
Tipo	NUMERIC	NUMERIC
Largo	12	12
Decimales	2	2
Signo	True	True

el mapping correspondiente es el siguiente:

```
private java.math.BigDecimal gxTv_SdtProduct_Productbase;
private java.math.BigDecimal gxTv_SdtProduct_Productprice;
```

En este caso, lo más interesante es que para no perder precisión al evaluar operaciones en Java se debe de utilizar el tipo de dato *BigDecimal*, y la forma de expresar operaciones implica utilizar métodos que representan dichas operaciones. En este caso, la traducción correcta de la expresión anterior es:

```
$p.setgxTv_SdtProduct_Productprice(
   $p.getgxTv_SdtProduct_Productprice()
      .add($p.getgxTv_SdtProduct_Productbase()
      .multiply(new java.math.BigDecimal(0.01))));
```

El proceso de traducción fue realizado utilizando el árbol sintáctico (AST- Abstract Syntax Tree) [195] que representa las reglas de negocio a nivel GeneXus. El árbol es recorrido en forma top-down y genera la representación de la plataforma por cada producción que se procese.

Es necesario revisar este proceso debido a que se necesita mucho mas inteligencia a nivel de la plataforma para poder decidir en forma correcta cómo debe de generarse cada producción. Este es uno de los puntos que resuelve en forma transparente el generador de código ya que tiene el conocimiento para realizar dicha transformación hasta el último detalle sintáctico. La implementación hecha no es completa en este sentido debido a no conocer los mecanismos de generación que se utilizan y en particular la heurística asociada.

6.6. Trabajos Futuros

Este trabajo se enfocó fundamentalmente en comprender la propuesta de administración de reglas de negocio, y aplicar dichos conceptos para lograr aplicaciones capaces de adaptarse rápidamente a cambios en los requerimientos [167].

Se investigó la infraestructura base que proveen los motores de reglas y se realiza e implementa una propuesta para validar dicho enfoque en el ecosistema GeneXus. En el transcurso de la investigación se encontraron otras tecnologías que tienen una fuerte relación con el concepto de reglas de negocio.

A continuación se detalla una breve reseña de posibles extensiones y mejoras que se pueden realizar a partir de este trabajo.

6.6.1. Procesos de Negocio

Existe una convergencia con respecto a las reglas de negocio y la administración de procesos. En la medida que ciertas partes de los procesos de negocio se puedan generalizar y representar como reglas de negocio, se logra una generalidad importante a nivel de procesos, expresando puntos de variabilidad en términos de reglas de negocio.

Posteriormente, los expertos del negocio pueden realizar modificaciones en dichas reglas, por lo que si las mismas se utilizan en el contexto de BPM (Business Process Management) [139], están facilitando la modificación tanto de los procesos como de las reglas.

La mayoría de los productos de BPM han incorporado tecnologías de evaluación de reglas, ya sea mediante desarrollo propio o con alianzas estratégicas con diferentes productos especializados y ofrecen dichas capacidades en forma transparente[1].

En la fase de evaluación, se realiza un caso de uso específico para validar dicho concepto y realiza una primera aproximación de cómo podría realizarse una interacción entre GXFlow[2] (administrador de procesos GeneXus) y el prototipo que administra reglas de negocio.

[1] `http://www.ebizq.net/white_papers/10571.html`
[2] `http://www.genexus.com/gxflow`

Se logra centralizar en reglas las decisiones claves que influyen sobre el flujo de proceso de ejemplo. Se podría extender el prototipo para investigar la interacción entre ambas tecnologías y lograr entender y resolver problemas existentes en procesos de negocios a través de cambios en reglas de negocio clave.

6.6.2. Procesamiento de Eventos

Están emergiendo soluciones que integran el enfoque de reglas de negocio con lo que se denomina CEP (Complex Event Processing) [175], una técnica que se encarga de procesar eventos en tiempo real.

Las reglas se definen utilizando los mismos constructores básicos ya definidos: términos y hechos. Se agrega la particularidad de que algunos de estos hechos pueden considerarse eventos. Los eventos vistos desde una perspectiva del procesamiento de reglas, es un hecho especial (inmutable) en donde se tiene una fuerte relación con el tiempo.

Se utiliza generalmente para detectar y responder a anomalías en el negocio, amenazas y oportunidades. Se trata de detectar y seleccionar aquellos eventos que son relevantes del conjunto de eventos que se generen, encontrando relaciones e infiriendo nuevo conocimiento a partir de dicha información.

Desde el punto de vista de herramientas existentes, muchos de los motores de evaluación de reglas están agregando funcionalidad para soportar el procesamiento de eventos; es decir, poder definir reglas que tengan en cuenta el concepto del tiempo y puedan reaccionar con el menor nivel de latencia al sucederse los eventos.

En el contexto del prototipo desarrollado, es interesante poder realizar pruebas de concepto que tengan en cuenta la temporalidad de las reglas y utilizar los mecanismos de evaluación para detectar y seleccionar aquellos eventos que sean relevantes, y así responder con las acciones que sean necesarias ejecutar.

6.6.3. Ontologías

El uso de ontologías [157] implica principalmente utilizar los siguientes servicios: *subsunción* donde se deriva información implícita a partir de una ontología y *satisfacibilidad* donde se realiza una clasificación de instancias con el objetivo de detectar inconsistencias en un modelo. Dichas tecnologías son funcionalmente inadecuadas desde el punto de vista de la administración de procesos y eventos de negocio. Esto incluye carencias en la representación de eventos, procesos, estados, acciones y otros conceptos. También hay importantes diferencias en lo que respecta a los procesos de inferencia utilizados por el enfoque ontológico y de evaluación de reglas. Los motores de evaluación de reglas asumen un universo cerrado [155], mientras que los modelos ontológicos asumen un universo abierto en el tratamiento del conocimiento.

Los motores de evaluación de reglas comerciales utilizan principalmente el enfoque de reglas de producción, en donde se enfatiza la acción sobre la lógica. Por ejemplo, esto se refleja en estándares como PRR (Production Rule Representation) [71] y RuleML [86].

Por otro lado, los estándares propuestos por la W3C [110], tales como OWL [68] y RIF (Rule Interchange Format) [82], se enfocan en el manejo de ontologías y lógica, pero no tienen en cuenta la acción. Algo similar ocurre con el estándar SBVR [93] impulsado por la OMG [57], donde se hace un énfasis en la parte lingüística de la definición de reglas de negocio, permaneciendo desconectada de los demás componentes de la solución.

El conocimiento cada vez más se representa en forma declarativa e independiente de la tecnología. Si bien actualmente no es posible intercambiar modelos semánticos entre diferentes proveedores ya que existe un gap importante entre lo que es la representación del conocimiento en términos ontológicos y reglas de negocio; la explotación de componentes que encapsulen conocimiento corporativo tiene un alto potencial y resulta natural pensar que en el mediano plazo ambos enfoques compartirán más puntos de contacto.

Utilizando ontologías es posible añadir semántica a los modelos por lo que permite ejecutar procesos de razonamiento mucho más completos. Existen varios proyectos (como por ejemplo ONTORULE [60]) trabajando para lograr combinar el uso de ontologías con reglas de producción y reglas lógicas. En el contexto GeneXus, resulta interesante investigar posibilidades en el procesamiento ontológico de una base de conocimiento con el objetivo de mejorar el proceso de captura y posterior tratamiento del conocimiento.

6.6.4. Separación de Intereses

En el prototipo realizado se logra separar efectivamente la definición de reglas del código de la aplicación que las ejecuta. Sin embargo, el código que llama a la evaluación de las reglas sigue estando diseminado en toda la aplicación. El desarrollador está obligado a adaptar o modificar el código que conecta la ejecución procedural de objetos generados con la evaluación de reglas cada vez que las reglas se deseen evaluar en diferentes eventos de disparo y contextos.

Un enfoque a investigar válido para lograr encapsular elegantemente este código es investigar la tecnología AOP (Aspect Oriented Programming – programación orientada a aspectos) [171], una tecnología específica que permite centralizar y encapsular código diseminado en un único módulo realizando una separación de intereses clara. Se podría investigar, en un trabajo futuro, una forma declarativa de conectar el concepto de reglas a los eventos de disparo en donde se desean aplicar, siendo esta tecnología una posible opción a evaluar.

También se podrían utilizar otros mecanismos, como por ejemplo la suscripción a eventos; es decir, publicar los diferentes eventos a los cuales los objetos GeneXus responden, y posteriormente idear un mecanismo que permita realizar modificaciones sobre las subscripciones. El objetivo buscado es modificar el comportamiento de un programa sin necesidad de acceder a su código fuente.

Otra tecnología a investigar, que también está relacionada, es lo que se denomina *Hook*:

> *In computer programming, hooking is a technique used to alter augment the behaviour of [a program], often without having access to its source code.*[3]

De esta manera, podría ser un mecanismo para permitir modificar comportamiento y componer e integrar diferentes aplicaciones. No solo permitiría realizar modificaciones a una aplicación en tiempo de ejecución, sino que también podría lograrse que otras aplicaciones ejecuten procesos en ciertos puntos en el flujo de ejecución sin necesidad de realizar modificaciones al código fuente.

6.6.5. Interfaz para el Experto del Dominio

El enfoque seguido en el desarrollo del prototipo para la administración de reglas de negocio implicó una administración textual del lenguaje de reglas en dos dimensiones: por un lado, lenguaje técnico para el desarrollador GeneXus, y un lenguaje "alto nivel" para el experto del negocio.

El primero se corresponde con el lenguaje diseñado GXRL (GeneXus Rule Language) representado por reglas en objetos de tipo *Ruleset*, mientras que el segundo se corresponde con la parametrización del lenguaje con el objetivo de lograr definiciones de reglas entendibles por parte del experto del dominio (objeto *RuleDSL*).

Si se compara con el enfoque MDA [191] promocionado por la OMG [57], se trabajó en los niveles de abstracción independiente del cálculo (CIM), independiente de la plataforma (PIM) y específico para una plataforma de ejecución (PSM).

A partir de la representación de alto nivel (objeto *RuleDSL* de la propuesta), se podría crecer mejorando la forma de representar el vocabulario relacionado al modelo del negocio (Business Object Model) y desarrollar interfaces más ricas para lograr un entendimiento mayor con los expertos de negocio, como son las tablas y árboles de decisión, flujos y otras formas de representar reglas de negocio.

En este sentido, resulta de particular interés el estándar SBVR [93] ya que está diseñado para definir reglas de negocio en lenguaje natural y podría jugar un rol protagónico en el acercamiento del experto del negocio con el personal informático. SBVR se encuentra definido en términos de lógica formal y tiene el detalle suficiente

[3] `http://webhooks.pbworks.com/`

para generar modelos y reglas en una plataforma específica. Se podría extender el lenguaje de reglas de alto nivel de abstracción propuesto en éste trabajo, utilizando un vocabulario que sea familiar al experto del negocio y lograr una representación de regla de negocio más potente y consistente.

6.6.6. Auditoria de Reglas

Un sistema de reglas de negocio siempre debe ser capaz de explicar el razonamiento por el cual llega a una conclusión o toma una acción tal como lo realizan las herramientas de workflow, en donde se puede tener acceso al histórico de la instancia de un proceso para conocer el recorrido de dicha instancia dentro del flujo de proceso. Es importante ofrecer una característica similar para la administración de reglas donde el sistema pueda responder con la agenda de ejecución de reglas, orden de disparo y acciones ejecutadas.

Si bien no se tuvo en cuenta dicha funcionalidad en el prototipo, se investigaron los mecanismos de suscripción a eventos de los motores de evaluación de reglas para "saber" qué reglas se ejecutan, siendo viable una implementación de este sentido.

6.6.7. Reglas Dinámicas versus Reglas Estáticas

El trabajo realizado se enfoca en la definición y ejecución de reglas de negocio que tengan la capacidad de modificar su comportamiento en tiempo de ejecución; es por eso que las denominamos reglas de negocio *dinámicas*. Sin embargo, en el contexto GeneXus ya existe el concepto de regla de negocio que podríamos denominar reglas de negocio *estáticas*, debido a que se resuelven en tiempo de generación, no teniendo capacidad de modificarse una vez que la aplicación se ejecuta.

El trabajo propone que las reglas de negocio se administren en forma centralizada como un objeto en sí mismo en la plataforma GeneXus, por lo que resulta importante estudiar cómo se podrían migrar las reglas de negocio estáticas a la representación de reglas propuesta y resolver un esquema en donde se integren en diseño y ejecución reglas dinámicas y estáticas.

6.6.8. Administración de Reglas

Al brindar mecanismos para definir y modificar restricciones del negocio en tiempo de ejecución, se torna fundamental proveer mecanismos para administrar correctamente el repositorio de reglas. Esto implica: versionado, administración en general, proveer seguridad y auditoria, entre otras tareas.

También es necesario mantener una trazabilidad completa sobre el ciclo de vida de una regla o juego de reglas que se administren en el repositorio, ya que es importante registrar todas las modificaciones, conocer qué usuarios realizan los cambios, verificar

y validar las modificaciones, así como también conocer cuándo se ingresan y se retiran reglas del ambiente de producción.

Esto permite que el experto de negocio tenga el control sobre las "reglas" que rigen las políticas y restricciones que a su vez inciden sobre los sistemas. A su vez implica que muchas de las tareas de aseguramiento de la calidad del software (Test, Validación y Verificación), que en general en el ciclo de desarrollo tradicional se resuelven a lo largo de todo el proceso de desarrollo, se deban reaplicar ante cualquier cambio que realice al juego de reglas a enviar a producción.

La capacidad de controlar las "reglas" y proveer mecanismos para poder validar cambios en tiempo de ejecución – antes de pasarlos a producción – permite aplicar este tipo de tecnología en escenarios de simulación comúnmente denominados *what-if* (que pasaría si...). El simular los resultados de la ejecución de un juego de reglas antes de su puesta en producción, le permitiría al experto del negocio "experimentar" con diferentes combinaciones de reglas para evaluar el impacto, de modo de poder arribar a la mejor solución posible.

Al conjunto de estas capacidades se le denomina sistema para gestión de reglas de negocio (BRMS - Business Rules Management System) [221]. Dichos componentes administran en forma completa el ciclo de vida de una regla; desde su concepción en lenguaje natural, modelado y almacenamiento en un repositorio de reglas, validación, verificación y posterior ejecución en un ambiente de producción. Resulta interesante extender el prototipo y lograr una plataforma en tiempo de ejecución que realice una administración completa de las reglas de negocio incorporando los puntos detallados anteriormente.

Además, en el contexto GeneXus sería interesante que el prototipo tenga capacidades para administrar la propagación de versiones de reglas entre la Base de Conocimiento y el entorno de ejecución (exportación e importación), proveer análisis de impacto, mecanismos de "merge" de reglas (para solucionar el problema que se produce cuando una regla se modifica tanto en el entorno de desarrollo como ejecución) y resolución de conflictos.

6.7. Reflexiones finales

Como reflexiones finales podemos decir que se lograron los objetivos planteados, utilizando la infraestructura base de la plataforma GeneXus, se desarrollaron nuevos artefactos para poder utilizar en forma satisfactoria el enfoque de administración de reglas de negocio.

Desde el punto de vista de modelado, estos artefactos se integran en forma transparente al ambiente de desarrollo, e interactúan con los componentes ya existentes de la plataforma logrando nuevas capacidades de representación y utilización de conocimiento.

En relación con el entorno de ejecución, las aplicaciones generadas aumentan significativamente sus capacidades de personalización y adaptabilidad, logrando que las empresas puedan ajustar sus procesos y reglas de negocio en respuesta a las condiciones de negocio cambiantes, administrando eficiente y efectivamente políticas y restricciones clave del negocio.

Glosario

Ad-hoc Ad-hoc se refiere a una solución elaborada específicamente para un problema o fin preciso y, por tanto, no es generalizable ni utilizable para otros propósitos. Se utiliza para referirse a algo que es adecuado sólo para un determinado fin. En sentido amplio, *ad-hoc* puede traducirse como *específico* o *específicamente*.

AI Se denomina Inteligencia Artificial (Artificial Intelligence[4]) a la rama de la ciencia informática dedicada al desarrollo de agentes racionales no vivos; entiéndase a un agente como cualquier artefacto capaz de percibir su entorno, es decir tener la capacidad de procesar tales percepciones y actuar en consecuencia.

API Interface de Programación de Aplicaciones (Application Programming Interface en inglés), es un conjunto de funciones, procedimientos, métodos que un sistema operativo, biblioteca o servicio provee para poder interactuar.

BAM Acrónimo de Business Activity Monitor; herramienta que permite monitorear servicios y procesos de negocio en la empresa, relacionado al concepto de BPM.

Base de Conocimiento Base de datos que contiene la metadata GeneXus[5].

BI Se denomina Inteligencia Empresarial o Inteligencia de Negocios (del inglés Business Intelligence) al conjunto de estrategias y herramientas enfocadas a la administración y creación de conocimiento mediante el análisis de datos existentes en una organización o empresa.

BNF Backus-Naur Form es una metasintaxis usada para expresar gramáticas libres de contexto, es un mecanismo para describir lenguajes formales.

BOM Acrónimo de Business Object Model, utilizado por la empresa IBM ILOG para representar el modelo de objetos de negocio.

Para que el BOM sea entendible por los analistas de negocio, se realiza una verbalización y se adjuntan los términos y frases correspondientes. A la lista completa de términos para escribir reglas se le denomina vocabulario.

BPMN Acrónimo de Business Process Modeling Notation; notación basada en un diagrama de flujos utilizado para definir procesos de negocio. Es el resultado de un acuerdo entre varios vendedores de herramientas de modelado.

[4] `http://www.grupoalianzaempresarial.com/inteligenciaartificial.htm`
[5] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?1836`

BPMS Acrónimo de Business Process Management System; sistema de administración de procesos de negocio.

BRM Acrónimo de Business Rule Management (Administración de reglas de negocio). Se trata de una disciplina estructurada que guía la definición, categorización, administración, puesta en producción y uso de reglas de negocio.

Ayuda a las organizaciones a estandarizar y optimizar sus reglas de negocio para obtener mayor visibilidad, centralización y consistencia en decisiones clave.

Business Component Business Component[6] es una propiedad de los objetos Transacción GeneXus que permite que dichos objetos se ejecuten sin interfaz de usuario.

Business Rule Mantra El término *mantra* en el contexto de la administración de reglas de negocios se define como:

> *"las reglas están basadas en hechos, y los hechos están basados en términos."*

Las reglas se construyen en base a hechos que a su vez se construyen a partir de conceptos expresados por términos. Los términos expresan conceptos del negocio; los hechos definen verdades sobre dichos conceptos.

Cláusula de Horn Una cláusula de Horn[7] es una regla de inferencia lógica con una serie de premisas (cero, una o más), y un único consecuente.

Prolog es uno de los lenguajes de programación lógica más conocidos que soporta cláusulas de Horn.

Conocimiento Explícito El conocimiento explícito (*Explicit Knowledge*) generalmente se evidencia en cualquier tipo de documentación, libros, fórmulas, contratos, diagramas de proceso, casos de estudio, manuales, etc. Es tangible, visible y puede ser accedido por cualquier tercero.

Conocimiento Táctico El conocimiento táctico (*Tacit Knowledge*) generalmente se encuentra en interacciones entre empleados y clientes; es difícil de catalogar, altamente experimental, difícil de documentar en detalle, efímero, intangible y transitorio.

CQL Acrónimo de Continuous Query Language; un lenguaje declarativo basado en SQL para registrar consultas en forma continua contra streams de información, utilizado en lo que se conoce como procesamiento de eventos complejos (CEP – Complex Event Processing) [175]. Extiende al cálculo relacional para incluir el concepto de evento.

[6] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?5846`
[7] `http://planetmath.org/?op=getobj&from=objects&id=8641`

DLL Acrónimo de Dynamic Linking Library (Bibliotecas de Enlace Dinámico); término con el que se refiere a los archivos con código ejecutable que se cargan bajo demanda del programa por parte del sistema operativo.

DSM Acrónimo de Domain Specific Modeling [169]; enfoque de desarrollo de sistemas utilizando modelos para guiar el proceso de análisis, diseño, construcción, distribución y mantenimiento.

DSL Acrónimo de Domain Specific Language [169]; lenguaje de programación diseñado para una tarea o propósito específico; en contraste con los lenguajes de programación de propósito general. Algunos ejemplos de uso de DSL son:

- Formulas definidas en una hoja de cálculo y macros.
- Gramática YACC[8] para la creación de parsers.
- Expresiones regulares.

ebXML Acrónimo de Electronic Business using eXtensible Markup Language[9], lenguaje para el intercambio y procesamiento de información a través de la Web.

Eclipse IDE de código abierto utilizado fundamentalmente en Java.

ECA Acrónimo de Event Condition Action (ECA) [118]; se utiliza para referirse a la estructura de reglas activas en una arquitectura orientada a eventos. Está compuesto de los siguientes elementos:

- *Evento*: especifica la señal que dispara la invocación de una regla.
- *Condición*: test lógico.
- *Acción*: actualización de datos o invocación a otros eventos.

EDA Acrónimo de Event Driven Architecture (EDA) [162]; patrón de arquitectura de software que promueve la producción, detección, consumo y reacción a eventos.

Este patrón arquitectónico se puede aplicar en diseño e implementación de aplicaciones y sistemas que necesiten transmitir eventos entre componentes de software con bajo acoplamiento (agentes que emiten y consumen eventos) como puede ser un marco orientado a servicios.

Entidad Colección de atributos que describen un objeto (persona, lugar, cosa) o un evento de interés para el Modelo de Datos.

[8] `http://dinosaur.compilertools.net/yacc/`
[9] `http://www.ebxml.org/`

EPF Acrónimo de Eclipse Process Framework (EPF); proyecto open source que es administrado por la Eclipse Foundation[10]. Tiene como objetivo producir un framework de proceso de ingeniería de software personalizable, con herramientas, ejemplos de procesos y soporte de una amplia gama de proyectos y estilos de desarrollo.

ESP Acrónimo de Event Stream Processing (ESP) [145]; tecnología que asiste a la construcción de sistemas utilizando el enfoque EDA (Event Driven Architecture).

ESP procesa múltiples flujos de datos de diferentes eventos con el objetivo de identificar los eventos significativos dentro de esos flujos, utilizando técnicas como correlación de eventos y abstracción, jerarquización de eventos, relación entre eventos como por ejemplo causalidad y membrecía.

Extensión Es uno o varios módulos que brindan una nueva funcionalidad a la plataforma GeneXus. Estos módulos son archivos de tipo DLL que se instalan en un directorio especifico de la instalación de GeneXus.

Flujo de Reglas Se define como un flujo que define el orden en el cual el conjunto de reglas debería ser evaluado para implementar una determinada parte de un proceso de decisión de negocio. Permite expresar:

- Secuencia, indicando el orden a tener en cuenta.
- Paralelismo, tareas que pueden ejecutarse a la vez.
- Opcionalidad, ramas condicionales que sólo se ejecutan si se cumplen las condiciones.

Framework Un framework[11], en el desarrollo de software es una estructura de soporte definida en la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre otros componentes para ayudar a desarrollar y unificar los diferentes componentes de un proyecto.

Gartner Gartner[12], Inc. (NYSE: IT) es una consultora de Tecnologías de la Información.

GeneXus GeneXus [178, 25] es una herramienta inteligente, basada en conocimiento puro, que permite a las empresas estar a la vanguardia en las últimas tecnologías sin tener que aprender a usarlas; y les permite aumentar su productividad y concentrarse en el negocio al automatizar el diseño, desarrollo y mantenimiento de aplicaciones de negocio multiplataforma, que fácil y rápidamente pueden

[10] `http://www.eclipse.org/org/`
[11] `http://es.wikipedia.org/wiki/Framework`
[12] `http://www.gartner.com/it/about_gartner.jsp`

transformar los cambios de la realidad en oportunidades que permitan expandir el negocio y hacerlo más rentable.

Producida en Uruguay por la empresa Artech, su primera versión fue liberada al mercado para su comercialización en 1989. Sus creadores, Breogán Gonda y Nicolás Jodal, han sido galardonados con el Premio Nacional de Ingeniería en 1995 otorgado por el "Proyecto GeneXus".

Gramática Ciencia que estudia los elementos de una lengua y sus combinaciones.

IDE Acrónimo de IDE (Integrated Development Environment); aplicación que brinda facilidades a los programadores para el desarrollo de software.

Inferencia Acto de procesar o derivar una conclusión basado solamente en información que ya se conoce.

IT Acrónimo de Information Technology; Tecnologías de la Información.

Java Lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 90.

JavaBeans Modelo de componentes creado por Sun Microsystems para la construcción de aplicaciones en Java. Se encapsulan varios objetos en un único objeto, para hacer uso de un sólo objeto en lugar de varios más simples. Debe satisfacer las siguientes convenciones:

- Definir un constructor sin argumentos.
- Sus propiedades deben ser accesibles mediante métodos *get* y *set* que siguen una convención de nomenclatura estándar.

Janino Compilador Java que puede leer expresiones, bloques, clases, código fuente y generar bytecode Java a ser cargado y ejecutado directamente desde memoria.

Algunos proyectos que utilizan Janino son Drools, JINX, Groovy, Farrago, Scripteffect, Deimos.

Jenabeans Biblioteca que permite generar RDF a partir de objetos Java que soporten la especificación JavaBeans [38].

JSR-94 JavaTM Rule Engine API (JSR 94), desarrollado por la Java Community Process, define una API Java para interactuar con motores de reglas desde la plataforma Java. La iniciativa JSR-94[13] es un intento de unificar en una única API las diferentes opciones existentes en el mercado para trabajar con motores de evaluación de reglas de negocio. La misma provee un *mínimo denominador común* de las funcionalidades de los diferentes motores de reglas.

[13] `http://jcp.org/en/jsr/detail?id=94`

El estándar no tuvo el éxito deseado debido a que no se ofrece suficiente funcionalidad desde el punto de vista de la API en sí y los diferentes proveedores no hicieron modificaciones o las hicieron parcialmente a sus productos para soportar la nueva interfaz definida.

Lenguaje Se considera como un conjunto de oraciones, que usualmente es infinito y se forma con combinaciones de palabras del diccionario. Es necesario que esas combinaciones sean correctas (con respecto a la sintaxis) y tengan sentido (con respecto a la semántica).

Lógica de descripción Lógica de descripción también llamadas lógicas descriptivas (DL por Description Logics) son una familia de lenguajes de representación del conocimiento que pueden ser usados para representar conocimiento terminológico de un dominio de aplicación de una forma estructurada y formalmente bien comprendida.

Lógica de Primer Orden La lógica de primer orden (First Order Logic en inglés) [209] es una extensión de la lógica proposicional en donde se agrega el cuantificador universal y existencial. Esto significa que se pueden definir condiciones en las reglas que sean verdaderas para todos los hechos en la base de hechos, o que sea verdadero o falso para al menos un hecho.

Las sentencias lógicas se forman de constantes, variables y sus cuantificadores, predicados y conectores como por ejemplo operadores lógicos (AND, OR, NOT), implicación y equivalencia.

Lógica Difusa La lógica difusa (del inglés fuzzy logic) [211] forma parte de la lógica multivaluada en donde se admiten varios valores de verdad posibles; se usan modelos matemáticos para representar nociones subjetivas, como *caliente – tibio – frío*, para valores concretos que puedan ser manipuladas por los ordenadores. En la lógica clásica una proposición sólo admite dos valores: *verdadero* o *falso*.

Lógica Proposicional La lógica proposicional (Propositional Logic en inglés) es una rama de la lógica clásica que estudia las proposiciones o sentencias lógicas, sus posibles evaluaciones de verdad y en el caso ideal, su nivel absoluto de verdad.

MDA Acrónimo de Model-Driven Architecture [191]; enfoque de diseño de software lanzado por la OMG en 2001.

Provee un conjunto de guías para expresar las especificaciones de un sistema como un conjunto de modelos. Define la funcionalidad del sistema utilizando una modelo independiente de la plataforma utilizando un lenguaje apropiado al dominio como por ejemplo CORBA, .NET, Web.

MOF Acrónimo de Meta-Object Facility[14]; estándar de la OMG para construir metamodelos.

[14] `http://www.omg.org/mof/`

MOF juega el mismo rol que EBNF cuando se define la gramática de un lenguaje de programación. MOF es un DSL utilizado para definir metamodelos.

MYCIN Sistema Experto para la realización de diagnósticos, iniciado por Ed Feigenbaum en la universidad de Standford y posteriormente desarrollados por E.Shortliffe y colaboradores.

OCL Acrónimo de Object Constraint Language, (OCL) [56]; lenguaje para la descripción formal de expresiones en los modelos UML.

Sus expresiones pueden representar invariantes, precondiciones, post-condiciones, inicializaciones, reglas de derivación, así como consultas a objetos para determinar su estado.

OMG El Object Management Group (OMG) [57] tiene como objetivo la creación de especificaciones estándar para la interoperabilidad, portabilidad y reutilización de aplicaciones empresariales en entornos distribuidos y heterogéneos. Es un consorcio industrial con más de 800 empresas, organizaciones e individuos; sin fines de lucro y de membresía abierta.

Ontología El término ontología [157] en informática hace referencia a la formulación de un exhaustivo y riguroso esquema conceptual dentro de un dominio dado, con la finalidad de facilitar la comunicación y permitir compartir información entre diferentes sistemas.

OWL Acrónimo de Ontology Web Language; lenguaje de etiquetas para publicar y compartir datos usando ontologías en la web. OWL tiene como objetivo facilitar un modelo de marcado construido sobre RDF y codificado en XML.

PIM PIM, Platform Independent Model [191]; modelo independiente de la plataforma del enfoque de desarrollo de software utilizando MDA.

Plataforma .Net Plataforma de desarrollo de software respaldada y desarrollada por Microsoft.

PMML Acrónimo de Predictive Model Markup Language[15]; lenguaje basado en XML que provee un mecanismo para que las aplicaciones definan modelos de data mining y estadísticos; para que se puedan intercambiar entre diferentes aplicaciones.

Paradigma de Programación Un paradigma de programación representa un enfoque particular o filosofía para la construcción de software. Tipos de paradigma:

- *Imperativo o por procedimientos*: considerado el más común y está representado por lenguajes como C o BASIC.

[15] `http://www.dmg.org/v4-0/GeneralStructure.html`

- *Declarativo*, por ejemplo SQL.
- *Funcional*: representado por la familia de lenguajes LISP, ML o Haskell.
- *Lógico*: representado por el lenguaje Prolog.
- *Orientado a objetos*: representado por lenguajes como Smalltalk, C#, Java.

POJO Acrónimo de Plain Old Java Object[16]; se refiere a tener clases simple Java que no dependan de ningún framework.

Política Cada organización es responsable de definir las políticas que gobiernan su negocio; en este contexto una política es una guía que gobierna las decisiones de negocio.

Una política puede requerir cientos incluso miles de reglas, que cambian a lo largo del tiempo. El proceso por el cual una empresa administra los cambios en las políticas se le denomina ciclo de vida de las reglas de negocio.

Programación Declarativa Un programa es declarativo, si describe las características de "algo" en vez de definir como crearlo. Las sentencias se declaran sin tener en cuenta ningún flujo, simplemente se define que hacer sin definir cómo hacerlo.

Programación Funcional La programación funcional[17] es un paradigma de programación declarativa basado en la utilización de funciones matemáticas.

Programación Imperativa En la programación imperativa también denominada procedural; se debe especificar qué se desea obtener (utilizando sentencias de código) y cómo se desea realizarlo (utilizando estructuras de control).

Los programas imperativos son un conjunto de instrucciones que le indican al computador cómo realizar una tarea.

Programación Lógica En los lenguajes de programación lógica, los programas consisten de un conjunto de sentencias que especifican que un objetivo deseado es verdadero. Esto difiere con la programación imperativa en donde los programas especifican un conjunto de sentencias que se deben de ejecutar para obtener el objetivo deseado.

No existen sentencias de control como por ejemplo *If-then-else* en este tipo de lenguajes lo que lo hace un lenguaje sumamente potente y difícil a la vez. Está representado por ejemplo por el lenguaje Prolog; a su vez surgen modelos más simples que soportan un subconjunto de elementos del paradigma como por ejemplo Lógica de Primer Orden.

[16] `http://www.martinfowler.com/bliki/POJO.html`
[17] `http://www.defmacro.org/ramblings/fp.html`

Programación orientada a Objetos La Programación orientada a objetos (Object-Oriented Programming[18] en inglés) es un paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones y programas.

Está basado en varias técnicas, incluyendo herencia, modularidad, polimorfismo, y encapsulamiento.

Su uso se popularizó a principios de la década de 1990; algunos ejemplos de lenguajes de programación que soportan la orientación a objetos son: Java, C#, Smalltalk.

Prolog Prolog es un lenguaje de programación lógico declarativo. La lógica del programa se expresa en términos de relaciones, y la ejecución es gatillada por el disparo de consultas sobre dichas relaciones.

Se basa en una implementación procedural de las cláusulas de Horn.

PSM Platform Specific Model, modelo específico para la plataforma, utilizado en el contexto MDA.

Python Python[19] es un lenguaje de programación interpretado creado por Guido van Rossum en el año 1990 desarrollado como un proyecto de código abierto, administrado por la Python Software Foundation[20].

R2ML REWERSE Rule Markup Language [75] es un lenguaje de reglas desarrollado por el grupo REWERSE con el objetivo de tener un mecanismo de intercambio de reglas entre diferentes herramientas.

RDF Acrónimo de Resource Description Framework (Marco de Descripción de Recursos[21]); framework para metadata en la web, desarrollado por el W3C.

Este modelo se basa en la idea de convertir las declaraciones de los recursos en expresiones con la forma *sujeto-predicado-objeto* (conocidas en términos RDF como tripletas). El *sujeto* es el recurso, es decir aquello que se está describiendo. El *predicado* es la propiedad o relación que se desea establecer acerca del recurso. Por último, el *objeto* es el valor de la propiedad o el otro recurso con el que se establece la relación. La combinación de RDF con otras herramientas como RDF Schema y OWL permite añadir significado y es una de las tecnologías esenciales de la Web semántica.

RDF Schema RDFS o Esquema RDF[22] es una extensión semántica de RDF. Un lenguaje primitivo de ontologías que proporciona los elementos básicos para la descripción de vocabularios publicada en abril de 1998 por la W3C.

[18] `http://java.sun.com/docs/books/tutorial/java/concepts/index.html`
[19] `http://www.python.org/about/`
[20] `http://www.python.org/psf/`
[21] `http://www.w3.org/2010/02/rdfa/`
[22] `http://www.w3.org/TR/rdf-schema/`

REA Acrónimo de Resources Events Actors[23]; método para el modelado de procesos de negocio.

Ruby Lenguaje de programación interpretado, reflexivo y orientado a objetos, creado por el programador japonés Yukihiro "Matz" Matsumoto, quien comenzó a trabajar en Ruby[24] en 1993, y lo presentó públicamente en 1995. Su implementación oficial es distribuida bajo una licencia de software libre.

RuleSet Las reglas de negocio se agrupan en lo que se denomina conjunto de reglas; define la secuencia en las cuales las reglas se tienen que aplicar para lograr una decisión.

RUP Acrónimo de Rational Unified Process[25]; framework de desarrollo de software iterativo creado por Rational Software Corporation, una división de IBM desde 2003. Es un conjunto de metodologías adaptables al contexto y necesidades de cada organización.

SDT Structured Data Type u Objeto Estructurado[26], es la representación GeneXus de cualquier estructura compleja.

Silogismo El silogismo es una forma de razonamiento deductivo que consta de dos proposiciones como premisas y otra como conclusión, siendo la última una inferencia necesariamente deductiva de las otras dos.

SLD Acrónimo de Selective Linear Definite clause resolution [216]; regla de inferencia básica utilizada en programación lógica.

SOA Acrónimo de Software Oriented Architecture (Arquitectura orientada a servicios) [134]; estilo arquitectónico cuyo objetivo es asegurar bajo acoplamiento entre agentes de software que interactúan.

SPARQL Lenguaje de consulta para obtener información de grafos RDF utilizado fundamentalmente en el contexto de la Web Semántica.

SQL Acrónimo de Structures Query Language; lenguaje de consulta estructurado para acceder a bases de datos relacionales.

Stakeholder Interesado de una manera u otra en un proyecto de software.

Strelka Herramienta visual [177] para especificar reglas en formato URML [176].

Test Unitario Método para asegurarse que unidades individuales de código funcionan correctamente; conforma la parte más pequeña de una aplicación que se puede testear.

[23] `http://reatechnology.com/what-is-rea.html`
[24] `http://www.ruby-lang.org`
[25] `http://www-01.ibm.com/software/awdtools/rup/`
[26] `http://wiki.gxtechnical.com/commwiki/servlet/hwikibypageid?6286`

Transacción (GeneXus) Objeto GeneXus que brinda información de los datos de la aplicación y cómo el usuario va a acceder a la aplicación para realizar el CRUD (insertar, leer, modificar o borrar datos). GeneXus a partir de las transacciones construye la Base de Datos, normalizada a tercera forma normal, y también genera programas para hacer las operaciones mencionadas.

Turing Complete La completitud de Turing se atribuye a máquinas físicas o lenguajes de programación que podrían ser universales si tuvieran almacenamiento infinito y fueran absolutamente fiables.

Alan Turing construyó un modelo formal de computador, la máquina de Turing, y demostró que existían problemas que una máquina no podía resolver.

La máquina de Turing es un modelo matemático abstracto que formaliza el concepto de algoritmo. Que sea equivalente a una máquina de Turing universal (UTM) esencialmente significa que es capaz de realizar cualquier tarea computacional, aunque no significa que lo pueda hacer en forma eficiente, rápido o fácil.

Un motor de evaluación de reglas es una UTM, si se tuviera memoria infinita se puede concluir que es Turing Complete.

UML Acrónimo de Unified Modeling Language (Lenguaje Unificado de Modelado) [103], lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; está respaldado por la OMG [103]. Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software.

Unidad de Ejecución Los motores de reglas se disponibilizan mediante lo que se denomina unidades de ejecución (eXecution Unit en inglés por sus siglas XU). Una unidad de ejecución es un adaptador que carga un conjunto de reglas y pasa datos entre la aplicación y el motor de ejecución; se pueden crear varias instancias de ejecución y dejarlas temporalmente en un pool para su posterior utilización.

URML Acrónimo de UML-Based Rule Modeling Language [176]; lenguaje basado en UML muy cercano a R2ML para la representación de reglas de negocio desarrollado por el grupo REWERSE. Su objetivo principal es proveer una forma constructiva visual de reglas.

Visual Studio IDE desarrollado por Microsoft para trabajar con la plataforma .Net.

Vocabulario Verbalización formal muy cercano al lenguaje natural de los conceptos y elementos del modelo de negocio.

W3C El consorcio World Wide Web [110] es una comunidad internacional para el desarrollo de estándares Web.

WCF Acrónimo de Windows Communication Foundation[27]; plataforma de mensajería que forma parte de la API de la Plataforma .NET 3.0. Fue creado con el fin de permitir una programación rápida de sistemas distribuidos y el desarrollo de aplicaciones basadas en arquitecturas orientadas a servicios.

WebPanel (GeneXus) Objeto GeneXus utilizado para definir páginas Web.

Web Semántica La Web Semántica[28] (del inglés Semantic Web) se basa en la idea de añadir metadatos semánticos a la World Wide Web. Estas características – que describen el contenido, el significado y la relación de los datos – se deben proporcionar de manera formal para que así sea posible evaluarlas automáticamente por máquinas de procesamiento. El objetivo es mejorar Internet ampliando la interoperabilidad entre los sistemas informáticos y reducir la necesaria mediación de operadores humanos.

Web Service Un servicio web es un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos en redes como por ejemplo Internet.

Workflow Define la secuencia (flujo) en la cual se debe de realizar el trabajo para completar un proceso.

XBRL Acrónimo de eXtensible Business Reporting Language[29]; lenguaje basado en XML para el intercambio de información financiera sobre las empresas.

xUnit Agrupa a varios frameworks de test orientado al código, denominados colectivamente como xUnit[30].

XMI Acrónimo de XML Metadata Interchange[31]; especificación para el Intercambio de metadata vía XML.

Puede ser utilizada para intercambiar cualquier metamodelo cuya metadata pueda ser expresada utilizando el estándar MOF; el uso más común es como formato de intercambio entre herramientas de modelado utilizando modelos UML.

XML Acrónimo de eXtensible Markup Language [18]; estándar creado por la W3C partiendo de las especificaciones de SGML[32].

[27] `http://msdn2.microsoft.com/en-us/netframework/aa663324.aspx`
[28] `http://semanticweb.org`
[29] `http://www.xbrl.org/`
[30] `http://www.opensourcetesting.org/`
[31] `http://www.omg.org/technology/documents/formal/xmi.htm`
[32] `http://www.w3.org/MarkUp/SGML/`

XML Schema XML Schema[33] es un lenguaje de esquema utilizado para describir la estructura y las restricciones de los contenidos de los documentos XML de una forma muy precisa, más allá de las normas sintácticas impuestas por el propio lenguaje XML. Se consigue así, una percepción del tipo de documento con un nivel de abstracción alto. Fue desarrollado por la W3C y alcanzó el nivel de recomendación en mayo de 2001.

XOM Acrónimo de eXecution Object Model (o eXecutable Object Model). El modelo de negocio debe de mapearse a un modelo físico de datos, a este modelo de datos se le denomina XOM. Cada elemento definido en el BOM debe tener su correspondiente en el modelo de implementación XOM, no necesariamente debe existir una correspondencia uno a uno entre ambos conceptos.

Este concepto es manejado en forma específica por la empresa ILOG; el modelo XOM puede implementarse como objetos Java, .Net o XML Schema.

XPDL Acrónimo de XML Process Definition Language[34]; define un formato XML para el intercambio de diagramas entre las distintas herramientas de modelado de procesos.

[33] `http://www.w3.org/XML/Schema`
[34] `http://wfmc.org/xpdl.html`

Bibliografía

[1] AAAI - Association for the Advancement of Artificial Intelligence. http://www.aaai.org/home.html. [Acceso 27/11/2009].

[2] Agile Alliance. http://www.agilealliance.org/. [Acceso 15/11/2009].

[3] Bosch drives into the rule engine market. http://bizrules.info/weblog/2008/09/bosch_drives_into_the_rule_eng.html. [Acceso 20/02/2010].

[4] BPMInstitute. http://www.bpminstitute.org/. [Acceso 27/11/2009].

[5] BRXG - Business Rule Experts Group. http://brexperts.ning.com/. [Acceso 27/11/2009].

[6] Business Rule Community. http://www.brcommunity.com/. [Acceso 27/11/2009].

[7] Business Rules for Electronic Commerce. http://www.research.ibm.com/rules/. [Acceso 15/02/2010].

[8] Business Rules Forum. http://www.businessrulesforum.com. [Acceso 27/11/2009].

[9] Business Rules Group. http://www.bpminstitute.org/. [Acceso 27/11/2009].

[10] CA Aion® Business Rules Expert. http://www.ca.com/us/products/product.aspx?id=250. [Acceso 15/02/2010].

[11] CLIPS: A Tool for Building Expert Systems. http://clipsrules.sourceforge.net/. [Acceso 15/02/2010].

[12] Corticon Studio. http://www.corticon.com/Products/Business-Rules-Modeling-Studio.php. [Acceso 15/02/2010].

[13] Drools. http://labs.jboss.com/drools. [Acceso 15/11/2009].

[14] Drools.NET. http://droolsdotnet.codehaus.org/. [Acceso 15/11/2009].

[15] EDM Summit. http://www.edmsummit.com. [Acceso 27/11/2009].

[16] Expergent - A RETE/UL based Business Rules Engine for the .net platform. http://code.google.com/p/expergent/. [Acceso 15/02/2010].

[17] Experian Decision Analytics. http://www.experian-da.com/. [Acceso 16/02/2010].

[18] Extensible Markup Language (XML). http://www.w3.org/XML/. [Acceso 15/11/2009].

[19] Fair Isaac Acquires Rulespower Technology To Advance Business Rules Capabilities. http://tinyurl.com/FairIsaacAcquiresRulesPower. [Acceso 20/02/2010].

[20] Fast-Growing RuleBurst Re-brands as Haley Limited, Expands U.S. Market Presence. http://www.businesswire.com/news/google/20080310005150/en/Fast-Growing-RuleBurst-Re-brands-Haley-Limited-Expands-U.S. [Acceso 20/02/2010].

[21] FICO™ Blaze Advisor® business rules management. http://www.fico.com/en/Products/DMTools/Pages/FICO-Blaze-Advisor-System.aspx. [Acceso 15/02/2010].

[22] Forrester Research. http://www.forrester.com. [Acceso 04/03/2010].

[23] Fuzzy Extension to the CLIPS Expert System Shell (FuzzyCLIPS). http://www.nrc-cnrc.gc.ca/eng/projects/iit/fuzzy-reasoning.html. [Acceso 15/02/2010].

[24] Gartner Technology Business Research Insight. http://www.gartner.com/. [Acceso 04/03/2010].

[25] GeneXus. http://www.genexus.com. [Acceso 16/02/2010].

[26] GOLD Parser - Morozov C# Engine. http://www.devincook.com/goldparser/engine/c-sharp/morozov/doc/index.html. [Acceso 16/02/2010].

[27] GOLD Parsing System - A Free, Multi-Programming Language, Parser Generator. http://www.devincook.com/goldparser/. [Acceso 16/02/2010].

[28] Haley Business Rules Engine. http://www.haley.com/products/haley-rule-services/business-rules-engine/index.html. [Acceso 15/02/2010].

[29] Hammurapi Rules. http://wiki.hammurapi.biz/index.php?title=Hammurapi_rules. [Acceso 15/11/2009].

[30] IBM to acquire ILOG in $340 million deal. http://news.cnet.com/8301-1001_3-10000450-92.html. [Acceso 20/02/2010].

[31] IDC Home: The premier global market intelligence firm. http://www.idc.com/. [Acceso 04/03/2010].

[32] InRule. http://www.inrule.com/. [Acceso 15/02/2010].

[33] ISWC - International Semantic Web Conference. http://iswc.semanticweb.org/. [Acceso 27/11/2009].

[34] Jade - Java Agent DEvelopment framework. http://jade.tilab.com/. [Acceso 15/11/2009].

[35] Java 2 Platform, Enterprise Edition (J2EE) Overview. http://java.sun.com/j2ee/overview.html. [Acceso 15/11/2009].

[36] Java Rule Engine API (JSR 94). http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html. [Acceso 16/11/2009].

[37] Java SE Overview. http://java.sun.com/javase/index.jsp. [Acceso 15/11/2009].

[38] Javabeans spec. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html. [Acceso 27/02/2010].

[39] JBoss, Inc. Reports Fourth Quarter Results. http://findarticles.com/p/articles/mi_m0EIN/is_2006_Feb_22/ai_n26768822. [Acceso 20/02/2010].

[40] Jena A Semantic Web Framework for Java. http://jena.sourceforge.net/. [Acceso 15/02/2010].

[41] Jena 2 Inference support. http://jena.sourceforge.net/inference/. [Acceso 15/02/2010].

[42] jenabean - a library for persisting java beans to rdf. http://code.google.com/p/jenabean/. [Acceso 27/02/2010].

[43] JEOPS - Java based Rule Engine. http://sourceforge.net/projects/jeops/. [Acceso 15/11/2009].

[44] Jess, the Rule Engine for the Java™ Platform.

[45] JLisa. http://jlisa.sourceforge.net/. [Acceso 15/11/2009].

[46] JRule Engine. http://jruleengine.sourceforge.net/. [Acceso 15/11/2009].

[47] JSONRules: A JavaScript-based rule engine. http://code.google.com/p/jsonrules/. [Acceso 15/02/2010].

[48] JxBRE - Java Business Rules Engine. http://sourceforge.net/projects/jxbre/. [Acceso 15/02/2010].

[49] Khammurabi Rule - Inference Engine. http://rubyforge.org/projects/khammurabi/. [Acceso 15/02/2010].

[50] Lab4Jefis: The Java Expert Fuzzy Inference System (JEFIS). http://www.lab4inf.fh-muenster.de/lab4inf/Lab4Jefis/project-summary.html. [Acceso 06/12/2009].

[51] Mandarax. http://sourceforge.net/projects/mandarax. [Acceso 15/11/2009].

[52] Markov algorithm. http://planetmath.org/encyclopedia/MarkovAlgorithm.html. [Acceso 22/02/2010].

[53] Microsoft Windows SharePoint Services 3.0. http://technet.microsoft.com/en-us/windowsserver/sharepoint/default.aspx. [Acceso 20/02/2010].

[54] Mvel. http://mvel.codehaus.org/. [Acceso 27/02/2010].

[55] .NET Business Rule Engine. http://nxbre.org/. [Acceso 15/02/2010].

[56] Object Constraint Language, Version 2.0. http://www.omg.org/spec/OCL/2.0. [Acceso 15/11/2009].

[57] Object Management Group. http://www.omg.org/. [Acceso 15/11/2009].

[58] October RulesFest. http://octoberrulesfest.org/. [Acceso 27/11/2009].

[59] onRules BRMS. http://www.delta-r.com/es/BRMS-onrules.html. [Acceso 15/02/2010].

[60] ONTORULE Project - Ontologies meets Business Rules. http://www.ontorule-project.eu/. [Acceso 15/11/2009].

[61] OpenLexicon.org. http://openlexicon.org/. [Acceso 15/11/2009].

[62] OpenRules. http://openrules.com/. [Acceso 15/11/2009].

[63] OpenUP Agile Business Rules Development Methodology. http://www.eclipse.org/epf/openup_component/openup_abrd.php. [Acceso 16/02/2010].

[64] Opsj. http://www.pst.com/opsj.htm. [Acceso 15/02/2010].

[65] Oracle Buys Haley. http://www.oracle.com/us/corporate/press/017631_EN. [Acceso 20/02/2010].

[66] Overview of Oracle Business Rules. http://download-uk.oracle.com/docs/cd/B25221_01/web.1013/b15986/intro.htm. [Acceso 15/02/2010].

[67] OWL 2 is now a Candidate Recommendation. http://www.w3.org/blog/SW/2009/06/15/owl_2_is_now_a_candidate_recommendation. [Acceso 20/02/2010].

[68] OWL Working Group - OWL. http://www.w3.org/2007/OWL. [Acceso 15/11/2009].

[69] PREDIGY Analytics Overview. https://www.firstdata.com/en_us/products/corporate-billers/data-and-analytics/analytics. [Acceso 16/02/2010].

[70] Prova language for rule based scripting of Java and agents, and knowledge and information integration. http://www.prova.ws/. [Acceso 15/02/2010].

[71] PRR 1.0. http://www.omg.org/spec/PRR/1.0/Beta1/PDF. [Acceso 15/11/2009].

[72] Pychinko: Rete-based RDF friendly rule engine. http://www.mindswap.org/~katz/pychinko/. [Acceso 15/02/2010].

[73] PyCLIPS Python Module. http://pyclips.sourceforge.net/web/. [Acceso 15/02/2010].

[74] pyrete - A Production Rule System implemented in Python. http://code.google.com/p/pyrete/. [Acceso 15/02/2010].

[75] R2ML - W3C RIF-WG Wiki. http://www.w3.org/2005/rules/wg/wiki/R2ML. [Acceso 20/02/2010].

[76] Red Hat Signs Definitive Agreement to Acquire JBoss. http://www.redhat.com/about/news/prarchive/2006/jboss.html. [Acceso 20/02/2010].

[77] REWERSE - Reasoning on the Web with Rules and Semantics. http://rewerse.net/. [Acceso 27/11/2009].

[78] RIF Basic Logic Dialect. http://www.w3.org/2005/rules/wiki/BLD. [Acceso 20/02/2010].

[79] RIF Core Dialect. http://www.w3.org/2005/rules/wiki/Core. [Acceso 20/02/2010].

[80] RIF is a W3C Candidate Recommendation. http://www.w3.org/blog/SW/2009/10/02/rif_is_a_w3c_candidate_recommendation. [Acceso 20/02/2010].

[81] RIF Production Rule Dialect. http://www.w3.org/2005/rules/wiki/PRD. [Acceso 20/02/2010].

[82] RIF Working Group - RIF. http://www.w3.org/2005/rules/. [Acceso 15/11/2009].

[83] Ruby Programming Language. http://www.ruby-lang.org/en/. [Acceso 15/11/2009].

[84] Ruby Rools: An Open-source Pure Ruby Rules-Engine. http://rools.rubyforge.org/. [Acceso 15/02/2010].

[85] Ruleby. http://ruleby.org/wiki/Ruleby. [Acceso 15/02/2010].

[86] RuleML - The Rule Markup Initiative. http://ruleml.org/. [Acceso 27/11/2009].

[87] Rules Engine from Pegasystems Inc. http://www.pega.com/Products/RulesTechnology.asp. [Acceso 15/02/2010].

[88] Rules Expo. http://www.rulesexpo.com/. [Acceso 27/11/2009].

[89] Rules Technology Summit. http://www.rulestechnologysummit.com/. [Acceso 27/11/2009].

[90] SAP Netweaver Business Rules Management. http://www.sap.com/platform/netweaver/components/brm/index.epx. [Acceso 15/02/2010].

[91] SAP to Acquire YASU Technologies and Add Leading Business Rules Management System to SAP NetWeaver®. http://www.sap.com/about/newsroom/press.epx?pressid=8434. [Acceso 20/02/2010].

[92] Savvion Announces Availability of Business Rules Management System. http://tinyurl.com/SavvionBRMS. [Acceso 20/02/2010].

[93] SBVR 1.0. http://www.omg.org/spec/SBVR/1.0/PDF. [Acceso 15/11/2009].

[94] SIE - Simple Inference Engine. http://homepage.ntlworld.com/peterhi/sie.html. [Acceso 15/02/2010].

[95] Simple Rule Engine. http://sourceforge.net/projects/sdsre. [Acceso 15/02/2010].

[96] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/. [Acceso 15/11/2009].

[97] SweetRules Project. http://sweetrules.projects.semwebcentral.org/. [Acceso 15/11/2009].

[98] SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.w3.org/Submission/SWRL/. [Acceso 15/11/2009].

[99] Take - A Java rule compiler. http://code.google.com/p/take/. [Acceso 15/11/2009].

[100] the Business Rule Group. http://www.businessrulesgroup.org/. [Acceso 15/02/2010].

[101] The Business Rules Framework. http://msdn.microsoft.com/en-us/library/aa561216(BTS.20).aspx. [Acceso 15/02/2010].

[102] Trilogy buying California public company for $3.3M. http://www.bizjournals.com/austin/stories/2005/12/05/daily19.html. [Acceso 20/02/2010].

[103] Unified Modeling Language. http://www.uml.org/. [Acceso 15/11/2009].

[104] USoft Rules: The Business Rules Company. http://www.usoft.com/. [Acceso 15/02/2010].

[105] Visual Rules. http://www.visual-rules.com/. [Acceso 15/02/2010].

[106] VORTE - Vocabularies, Ontologies and Rules in the Enterprise. http://oxygen.informatik.tu-cottbus.de/VORTE/. [Acceso 27/11/2009].

[107] WebSphere ILOG Jrules. http://www-01.ibm.com/software/integration/business-rule-management/jrules/. [Acceso 15/02/2010].

[108] WebSphere ILOG Rules for .Net BRMS. http://www-01.ibm.com/software/integration/business-rule-management/rulesnet-family/. [Acceso 15/02/2010].

[109] Windows Workflow Foundation. http://msdn.microsoft.com/en-us/netframework/aa663328.aspx. [Acceso 15/02/2010].

[110] World Wide Web Consortium (W3C). http://www.w3.org. [Acceso 15/11/2009].

[111] Zilonis. http://www.zilonis.org/. [Acceso 15/11/2009].

[112] Abbas, N., Fayyaz, F., and Naeem, M. Business Rules in Software Development. Master's thesis, Lunds Universitet/Department of Informatics, 2008.

[113] Allen, J. *Anatomy of LISP*. McGraw-Hill, Inc., New York, NY, USA, 1978.

[114] Alvarez, Borriero, Valenzani, and Oliveri. Entorno de desarrollo Web para Genexus. Tech. rep., Facultad de Ingeniería - Universidad ORT, 2009.

[115] Arias, F. J., Moreno, J., and Ovalle, D. A. Integración de Mecanismos de Razonamiento en Agentes de Software Inteligentes para la Negociación de Energía Eléctrica.

[116] Arias, F. J., Moreno, J., and Ovalle, D. A. Integración de Ontologías y Capacidades de Razonamiento en Agentes de Software Inteligentes para la Simulación del Proceso de Negociación de Contratos de Energía Eléctrica. *Avances en Sistemas e Informática* (2007).

[117] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., Eds. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA, 2003.

[118] Bailey, J., Poulovassilis, A., and Wood, P. T. An event-condition-action language for XML. In *WWW '02: Proceedings of the 11th international conference on World Wide Web* (New York, NY, USA, 2002), ACM, pp. 486–495.

[119] Bajec, M., and Krisper, M. Issues and challenges in Business rule-based Information Systems. In *ECIS* (Regensburg, Germany, May 2005).

[120] Bajec, M., and Krisper, M. A methodology and tool support for managing business rules in organisations. *Information Systems 30*, 6 (sep 2005), 423–443.

[121] Bali, M. *Drools JBoss Rules 5.0 Developer's Guide*. PPackt Publishing Limited, 2009.

[122] Banaszewski, R. F. Paradigma Orientado a Notificaciones: Avances y Comparaciones. Master's thesis, Universidade Tecnológica Federal do Paraná, 2009.

[123] Barker, V. E., O'Connor, D. E., Bachant, J., and Soloway, E. Expert systems for configuration at Digital: XCON and beyond. *Commun. ACM 32*, 3 (1989), 298–318.

[124] Batory, D. The LEAPS Algorithm. Tech. rep., University of Texas, Austin, TX, USA, 1994.

[125] Belén, M., Blanco, P., and Sbrocca, L. Generador de Reglas de Negocio. Tech. rep., INCO - Facultad De Ingeniería, 2005.

[126] Berstel, B., Bonnard, P., Bry, F., Eckert, M., and Patranjan, P.-L. Reactive Rules on the Web. In *Reasoning Web* (2007), pp. 183–239.

[127] Bhoopalam, K. Fire - A Description Logic based Rule Engine for OWL Ontologies with SWRL-like Rules. Master's thesis, Concordia University Montreal, Canada, 2005.

[128] BIZRULES®. BRE Family Tree™ 2009. `http://bizrules.info/page/art_brefamilytree.htm`. [Acceso 20/02/2010].

[129] Boswell, W. The Top Ten Job Search Engines on the Web. `http://websearch.about.com/od/enginesanddirectories/tp/jobsearchengine.htm`. [Acceso 16/02/2010].

[130] Brant, D., Grose, T., Lofaso, B., and Miranker, D. Effects of Database Size on Rule System Performance:Five Case Studies. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB)* (1991).

[131] Browne, P. Financial knowledge capture using Red-Piranha: Rules, Workflow, Search and Enterprise Web 2.0. Master's thesis, University College Dublin, 2007.

[132] Buchanan, B. G., and Shortliffe, E. H. *Rule Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project (The Addison-Wesley series in artificial intelligence).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[133] Canedo, G., and Guridi, R. Ejecutor de Reglas de Negocio. Tech. rep., INCO - Facultad De Ingeniería, 2005.

[134] Carter, S. *The New Language of Business: SOA & Web 2.0.* IBM Press, 2007.

[135] Chisholm, M. *How to Build a Business Rules Engine: Extending Application Functionality through Metadata Engineering.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[136] Cibrán, M. A. *Connecting High-Level Business Rules with Object-Oriented Applications: An approach using Aspect-Oriented Programming and Model-Driven Engineering.* PhD thesis, Vrije Universiteit Brussel, 2007.

[137] Codd, E. F. A Relational Model of Data for Large Shared data Banks. *Commun. ACM 13*, 6 (1970), 377–387.

[138] Czarnecki, K., and Helsen, S. Classification of Model Transformation Approaches. In *OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture* (California, USA, October 2003).

[139] Debevoise, T. *Business Process Management With a Business Rules Approach: Implementing the Service Oriented Architecture.* BookSurge Publishing, 2007.

[140] Dechter, R., and Pearl, J. Generalized best-first search strategies and the optimality of A*. *J. ACM 32*, 3 (1985), 505–536.

[141] Doorenbos, R. B. *Production matching for large learning systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1995.

[142] Doyle, J. A truth maintenance system. 259–279.

[143] Emaldía, A., and Cristián, J. Criterios de selección para las herramientas de orquestación de servicios Web. Tech. rep., Facultad De Ingeniería - Universidad Nacional Mayor de San Marcos, 2008.

[144] Erl, T. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[145] Etzion, O. Semantic approach to event processing. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems* (New York, NY, USA, 2007), ACM, pp. 139–139.

[146] Feigenbaum, E. A., Buchanan, B. G., and Lederberg, J. On generality and problem solving: a case study using the DENDRAL program. Tech. rep., Stanford University, Stanford, CA, USA, 1970.

[147] Fernández, J. M. F. Generación automática de Sistemas Basados en Reglas mediante Programación Genética. Master's thesis, Facultad de Informática - Universidad Politécnica de Madrid, 2008.

[148] Fischer, K. The Rule-Based Multi-Agent System MAGSY. In *Keele University* (1993).

[149] Forgy, C. L. *On the efficient implementation of production systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1979.

[150] Forgy, C. L. Rete: a fast algorithm for the many pattern/many object pattern match problem. 324–341.

[151] Friedman-Hill, E. *Jess in Action: Java Rule-based Systems*. Manning Publications, 2003.

[152] Gálvez, J., Gómez, F. I., Guzmán, E., and Conejo, R. Un Sistema Inteligente para el Aprendizaje de Fundamentos de Programación Orientada a Objetos. In *Actas de la XII Conferencia de la Asociación Española para la Inteligencia Artificial* (2007), vol. 2, pp. 329–338.

[153] Garrigós, I., Gómez, J., and Cachero, C. Personalización de Aplicaciones en OO-H. Tech. rep., Departamento de Lenguajes y Sistemas Informáticos - Universidad de Alicante, 2002.

[154] Garrigós, I., Gómez, J., and Cachero, C. Modelado Conceptual de aplicaciones adaptivas y proactivas en OO-H. Tech. rep., Departamento de Lenguajes y Sistemas Informáticos - Universidad de Alicante, 2003.

[155] Giarratano, J. C., and Riley, G. *Expert Systems, Principles and Programming*. Course Technology, 2005.

[156] Gichahi, H. K. Rule-based Process Support for Enterprise Information Portal. Diplomarbeit, TU Hamburg-Harburg, Feb. 2003.

[157] Gruber, T. Ontología - Encyclopedia of Database Systems. http://tomgruber.org/writing/ontology-definition-2007.htm. [Acceso 23/03/2010].

[158] Hakizumwami, B. Building Enterprise Services with Drools Rule Engine. http://onjava.com/pub/a/onjava/2007/01/17/building-enterprise-services-with-drools-rule-engine.html. [Acceso 15/02/2010].

[159] Hay, D., and Healy, K. A. GUIDE Business Rules Project. Tech. rep., Chicago, USA, 1997.

[160] Hay, D., and Healy, K. A. Defining Business Rules - What Are They Really? Tech. rep., Business Rules Group, 2000.

[161] Herbst, H. Business Rules in Systems Analysis: a Meta-Model and Repository System. *Information Systems 21*, 2 (April 1996), 147–166.

[162] Ibrahim, M., and Etzion, O. Workshop on event driven architecture. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), ACM, pp. 624–624.

[163] Jackson, P. *Introduction to Expert Systems*. Addison-Wesley Professional, 1998.

[164] Jin, R., and Liang, Y. Implementation choice of business rules in e-commerce. Master's thesis, Lunds universitet/Department of Informatics, 2008.

[165] Joobbani, R., and Siewiorek, D. WEAVER: A Knowledge-Based Routing Expert. *IEEE Des. Test 3*, 1 (1986), 12–23.

[166] Kalp, D., Tambe, M., Gupta, A., Forgy, C., Newell, A., Acharya, A., Milnes, B., and Swedlow, K. Parallel OPS5 User's Manual. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.612, 1988.

[167] Kapocius, K., and Butleris, R. Business rules driven approach for elicitation of IS requirements. In *WMSCI 2005: the 9th world multi-conference on systemics, cybernetics and informatics* (Orlando, USA, May 2005).

[168] Kelly, M. A., and Seviora, R. E. An Evaluation of DRete on CUPlD for OPSS Matching. In *IJCAI* (1989), pp. 84–90.

[169] Kelly, S., and Tolvanen, J. P. *Domain-Specific Modeling*. John Wiley & Sons, 2007.

[170] Kent, S. Model Driven Engineering. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods* (London, UK, 2002), Springer-Verlag, pp. 286–298.

[171] Kiczales, G. Aspect-oriented programming. *ACM Comput. Surv.* (1996), 154.

[172] Krishnamoorthy, C. S. *Artificial Intelligence and Expert Systems for Engineers*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[173] Leannah, W. J. Exploring Business Application agility through Rules-based processing: Centralizing the Business Rules of an Enterprise. Master's thesis, Marquette University Wisconsin, 2006.

[174] Lee, P.-Y., and Cheng, A. M. K. HAL: A Faster Match Algorithm. *IEEE Trans. on Knowl. and Data Eng. 14*, 5 (2002), 1047–1058.

[175] Luckham, D. C. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[176] Lukichev, S. A UML-Based Rule Modeling Language (URML) on REWERSE Working Group, 2006. `http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/7`.

[177] Lukichev, S. Strelka - An URML-Based Visual Rule Modeling Tool, 2006. `hhttp://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/10`.

[178] Márquez Lisboa, D. *GeneXus - Desarrollo Basado en el Conocimiento*. Grupo Magró, 2006.

[179] Martín, F., Alonso, J., Hidalgo, M., and Ruiz, J. Razonamiento Automático en Sistemas de Representación del Conocimiento. Tech. rep., Departamento de Ciencias de la Computación e Inteligencia Artificial - Universidad de Sevilla, 1998.

[180] McCracken, D. D., and Reilly, E. D. Backus-Naur form (BNF). In *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., Chichester, UK, 2003, pp. 129–131.

[181] Miranker, D. P. *TREAT: a new and efficient match algorithm for AI production systems*. PhD thesis, Columbia University, New York, NY, USA, 1987.

[182] Mitchell, T. M. *Machine Learning*. McGraw-Hill, New York, 1997.

[183] Montes de Oca, I. M., Martínez del Busto, M. E., and González, L. Modelación orientada a hechos en el enfoque de reglas de negocio. Tech. rep., Informática en Salud, 2008.

[184] Morgan, T. *Business Rules and Information Systems: Aligning IT with Business Goals*. Addison-Wesley Professional, 2002.

[185] Moss, L. T., and Atre, S. *Business Intelligence Roadmap: The Complete Project Lifecycle for Decision-Support Applications*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[186] Muñoz Fernández, S. Procesamiento del lenguaje natural para recuperación de información. `http://procesamientolenguajerecuperacion.50webs.org/`. [Acceso 23/03/2010].

[187] Negnevitsky, M. *Artificial Intelligence: A Guide to Intelligent Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[188] Nicolae, O., Diaconescu, I.-M., Giurca, A., and Wgner, G. Sharing Rules between JBoss and Jena. *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on 0* (2007), 105–108.

[189] Nicolae, O., Giurca, A., and Wagner, G. On Interchange between Drools and Jess. In *Advances in Intelligent and Distributed Computing: Proceedings of the 1st International Symposium on Intelligent and Distributed Computing* (2007).

[190] Nordgren, J., and Bohman, C. Operation of and knowledge of Business Rules in Swede organisations. Master's thesis, Lund University, 2008.

[191] Object Management Group, Inc. *MDA Guide, v1.0.1*, June 2006. `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.

[192] Oguz, G. Decision Tree Learning for Drools. Master's thesis, Ecole Polytechnique Federale de Lausanne, 2008.

[193] Orr, K. T. *Data structured systems development methodology*. Ken Orr and associates, 1982.

[194] Ortín, M. J., Molina, J. G., Moros, B., and Nicolás, J. El Modelo de Negocio como Base del Modelo de Requisitos. Tech. rep., Facultad de Informática - Universidad Murcia, 2004.

[195] Parr, T. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[196] Paschke, A. RBSLA A declarative Rule-based Service Level Agreement Language based on RuleML. In *CIMCA '05: Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce Vol-2 (CIMCA-IAWTIC'06)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 308–314.

[197] Paschke, A. ECA-LP / ECA-RuleML: A Homogeneous Event-Condition-Action Logic Programming Language. *CoRR abs/cs/0609143* (2006).

[198] Paschke, A., Bichler, M., and Dietrich, J. Contractlog: An Approach to Rule Based Monitoring and Execution of Service Level Agreements. In *RuleML* (2005), pp. 209–217.

[199] Paschke, A., Dietrich, J., Giurca, A., Wagner, G., and Lukichev, S. On Self-Validating Rule Bases. In *Proceedings of 2nd International Workshop on Semantic Web Enabled Software Engineering, Athens, Georgia, USA (6th November 2006)* (2006).

[200] Paschke, A., Dietrich, J., and Kuhla, K. A Logic Based SLA Management Framework. Tech. rep., Technische Universitat Munchen, 2005.

[201] Pehla, M. Building a Distributed Rule Translator System as a Web Service. Tech. rep., Brandenburg University of Technology at Cottbus, Faculty of Computer Science, 2007.

[202] Perlin, M. The Match Box Algorithm for Parallel Production System Match. Tech. rep., Department of Computer Science, Carnegie Mellon University, 1989.

[203] Ross, R. G. *The Business Rule Book: Classifying, Defining and Modeling Rules.* Business Rule Solutions Inc, 1997.

[204] Ross, R. G. The Business Rule Approach. *Computer 36* (2003), 85–87.

[205] Ross, R. G. *Principles Of Business Rule Approach.* Aw Professional, 2003.

[206] Ross, R. G. *Business Rule Concepts - Getting to the Point of Knowledge.* Business Rule Solutions Inc, 2005.

[207] Roy, J., and Auger, A. Reasoning Processes, Methods and Systems for Use in Knowledge-Based Situation Analysis Support Systems. Tech. rep., Defence R&D Canada - Valcartier, Valcartier QUE (CAN), Canada, 2008.

[208] Rubio, E. V. *Incorporación de un sistema basado en reglas en un entorno de tiempo real.* Tesis doctoral en informática, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2004.

[209] Russell, S. J., and Norvig, P. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[210] Rymer, J. R. Trends: Business Rules Platforms 2008. http://www.forrester.com/rb/Research/trends_business_rules_platforms_2008/q/id/44374/t/2. [Acceso 20/02/2010].

[211] Siler, W., and Buckley, J. J. *Fuzzy Expert Systems and Fuzzy Reasoning*. Wiley-Interscience, 2004.

[212] Sinur, J. Business Rule Interchange: Fantasy or Fact of Life. http://blogs.gartner.com/jim_sinur/2008/12/29/business-rule-interchange-fantasy-or-fact-of-life/. [Acceso 20/02/2010].

[213] Smaizys, A., and Vasilecas, O. Business Rules Based Agile ERP Systems Development. *Informatica 20*, 3 (2009), 439–460.

[214] Sottara, D. Reasoning with Uncertainty in Drools, 2008. http://www.jboss.org/community/docs/DOC-9601.pdf;jsessionid=DE65361B9BE9881CAABA01659A20F431.

[215] Stefik, M. *Introduction to knowledge systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[216] Sterling, L., and Shapiro, E. *The art of Prolog: advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1986.

[217] Tadesse, T. Business Rules Object-Oriented Method [BROOM] for Business Rules System Development. Master's thesis, Addis Abada University, 2004.

[218] Taylor, J., and Raden, N. *Smart Enough Systems: How to Deliver Competitive Advantage by Automating Hidden Decisions*. Prentice Hall, 2007.

[219] Valverde, A. Gestión por Procesos usando mejora continua y reingeniería de procesos. Tech. rep., Facultad De Ingeniería - Universidad Nacional Mayor de San Marcos, 2007.

[220] Vincent, P. Business Rules Forum 2008: The BRE Vendor Panel. http://tibcoblogs.com/cep/2008/10/29/business-rules-forum-2008-the-bre-vendor-panel/. [Acceso 20/02/2010].

[221] von Halle, B. *Business Rules Applied-Business Better Systems, Using the Business Rules Approach*. John Wiley - Sons, 2002.

[222] von Halle, B. What is a Business Rules Approach? *The Data Administrator Newsletter* (2002).

[223] Wagner, G., Giurca, A., and Lukichev, S. A General Markup Framework for Integrity and Derivation Rules. In *Principles and Practices of Semantic Web Reasoning* (Dagstuhl, Germany, 2006), F. Bry, F. Fages, M. Marchiori, and H.-J. Ohlbach, Eds., no. 05371 in Dagstuhl Seminar Proceedings, Internationales Begegnungs - und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[224] Waltz, E. *Knowledge management in the intelligence enterprise.* Artech House, Boston, Mass., 2003.

[225] Weisstein, E. W. Tower of Hanoi. *MathWorld - A Wolfram Web Resource.*

[226] Winston, P. H. *Artificial Inlelligence.* Adison-Wesley Publishing Co, 1992.

[227] Zacharias, V. *Tool Support for Finding and Preventing Faults in Rule Bases.* PhD thesis, Universität Karlsruhe, 2008.

A. Business Rules Manifesto

El *Business Rule Manifesto* (extraído de `www.businessrulesgroup.org/brmanifesto.htm`) sintetiza el enfoque de reglas de negocio, estableciendo las principales ideas y características. Fue escrito por la *Business Rule Group* [100] con el objetivo de proveer una visión concisa sobre los principios fundamentales que sustenta al enfoque de reglas de negocio.

- **Artículo 1**. Los requerimientos como elementos principales, nunca como secundarios.
  1. Las reglas son un ciudadano de primera clase en el mundo de los requerimientos.
  2. Las reglas son esenciales para los modelos de negocio y para los modelos de tecnología.
- **Artículo 2**. Las Reglas son Independientes y no se encuentran contenidos en los procesos.
  1. Las reglas son restricciones explícitas de comportamiento y/o proporcionan soporte al comportamiento.
  2. Las reglas no son procesos ni procedimientos y por tanto no deben estar contenidas en ninguno de ellos.
  3. Las reglas se aplican a lo largo de los procesos y procedimientos. Debe existir un juego coherente de reglas que se aplique sistemáticamente en todas las áreas de actividad del negocio.
- **Artículo 3**. Proporcionar conocimiento meditado, no un sub-producto.
  1. Las reglas se construyen sobre hechos, y los hechos sobre conceptos tal y como son expresados mediante términos.
  2. Los términos expresan conceptos de negocio; los hechos realizan afirmaciones sobre estos conceptos; las reglas restringen y apoyan estos hechos.
  3. Las reglas deben ser explicitas. No se debe asumir ninguna regla sobre ningún concepto o hecho.
  4. Las reglas son los fundamentos que definen lo que el negocio sabe de sí mismo – es decir son el conocimiento básico del negocio.

5. Las reglas necesitan ser alimentadas, protegidas y gestionadas.

- **Artículo 4**. Declarativas, no procedurales.
  1. Las reglas deben expresarse de forma declarativa en sentencias de lenguaje natural, por la audiencia conocedora del negocio.
  2. Si algo no puede ser expresado claramente, entonces no es una Regla.
  3. Una serie de enunciados sólo es declarativa si no contiene una secuencia implícita.
  4. Cualquier enunciado de reglas que necesite de otros elementos que no sean términos o hechos, revelan hipótesis sobre la implementación de un sistema.
  5. Una regla es distinta del nivel de cumplimiento definido para ella. La regla y su nivel de cumplimiento son dos asuntos diferentes.
  6. Las reglas deben definirse independientemente de quién tiene la responsabilidad de su cumplimiento, de dónde, cuándo o cómo se deben cumplir.
  7. Las excepciones a las reglas se definen mediante otras reglas.
- **Artículo 5**. Expresiones bien formadas, no expresiones creadas con fines específicos (Ad-hoc).
  1. Las reglas de negocio se deben expresar de manera que pueda ser validada su exactitud por el personal conocedor del negocio.
  2. Las reglas de negocio se deben expresar de manera que se pueda verificar recíprocamente su coherencia.
  3. Las lógicas formales, como la lógica de predicados, son fundamentales para la expresión formal de reglas en términos de negocio, así como para las tecnologías que implementan dichas reglas.
- **Artículo 6**. Arquitectura basada en las reglas, no una implementación indirecta.
  1. Un sistema basado en reglas de negocio se construye intencionadamente para permitir el cambio continuo de las reglas de negocio. La plataforma sobre la que el sistema se ejecuta debe soportar esta evolución.
  2. Es mejor ejecutar las reglas directamente – por ejemplo utilizando un motor de reglas – antes que transcribirlas en alguna forma embebida dentro de un procedimiento.
  3. Un sistema de reglas de negocio siempre debe ser capaz de explicar el razonamiento por el cual llega a una conclusión o toma una acción.
  4. Las reglas se basan en los valores ciertos. La forma en la que certeza de una regla se determina, se mantiene oculta a quienes la utilizan.

5. La relación entre eventos y reglas es generalmente de muchos-a-muchos.

- **Artículo 7**. Procesos guiados por reglas, no programación basada en excepciones.
  1. Las reglas definen el límite entre actividad de negocio aceptable y no aceptable.
  2. Las reglas requieren a menudo de una gestión especial o específica de las violaciones detectadas. Cualquier actividad derivada de la violación de una regla es una actividad como cualquier otra.
  3. Para asegurar la máxima consistencia y reutilización, el tratamiento de las actividades de negocio no aceptables, debe separarse de la gestión de actividades de negocio aceptables.

- **Artículo 8**. Al servicio del negocio, no al de la tecnología.
  1. Las reglas tratan sobre las prácticas de la gestión y gobierno del negocio, por lo tanto son motivadas por las metas y los objetivos de negocio y se les da forma a través de varios factores internos y externos a la empresa.
  2. Las reglas suponen siempre un costo a la empresa.
  3. El costo de la aplicación de las reglas debe balancearse teniendo en cuenta los riesgos asumidos por el negocio, y las oportunidades perdidas en caso de no aplicarlas.
  4. "Más reglas" no es mejor, la abundancia de reglas no beneficia a su aplicación. Normalmente es mejor un número menor de reglas bien reflexionadas.
  5. Un sistema efectivo puede estar basado en un pequeño número de reglas. Adicionalmente, se pueden añadir reglas más discriminatorias, por las que a medida que pasa el tiempo el sistema mejore y se haga más inteligente.

- **Artículo 9**. "De, por y para" el personal de negocio. No "de, por y para" el personal de IT.
  1. Las reglas deben provenir del personal con conocimiento de negocio.
  2. Los expertos de negocio deben tener disponibles herramientas que les ayuden a formular, validar y gestionar reglas.
  3. Los expertos de negocio deben tener disponibles herramientas que les ayuden a verificar la coherencia recíproca entre las reglas de negocio.

- **Artículo 10**. Gestionando la lógica de negocio, no las plataformas de Hardware/Software.
  1. Las reglas de negocio son un patrimonio vital del negocio.

2. A largo plazo, las reglas son más importantes para el negocio que las plataformas de Hardware\Software.
3. Las reglas de negocio deben organizarse y salvaguardarse de forma que puedan ponerse en producción en nuevas plataformas de Hardware/Software.
4. Las reglas, y la habilidad para cambiarlas de forma eficaz, son factores clave para mejorar la adaptabilidad de las empresas.

B. Empresas y Herramientas

Existe una cantidad importante de herramientas y empresas en el área de administración de reglas de negocio. Se resume a continuación las principales empresas y sus respectivas herramientas.

B.1. Empresas

Empresas que están comprometidas con el enfoque de administración de reglas de negocio [204] se detallan a continuación.

Artemis Alliance

Artemis Alliance[1] es una empresa dedicada a la consultoría en lo que es la adopción del enfoque de reglas de negocio como metodología de desarrollo.

Corticon Technologies

Corticon Technologies[2] junto con *IBM ILOG* y *FICO* conforman los líderes de los motores de evaluación reglas de negocio.

La oficina central de *Corticon* se encuentra en Redwood, California, dispone de oficinas en Holanda y ha desarrollado una extensa red de partners. Su producto relacionado al área es *Corticon Studio*.

FICO

FICO[3] es una de las empresas que más peso tiene en el mercado de motores de evaluación de reglas y administradores de reglas de negocio. Su herramienta *Blaze Advisor* es utilizada (según análisis de la empresa Forrester) por más de 400 clientes en más de 500 instalaciones[4].

[1] `http://www.artemisalliance.com/`
[2] `http://www.corticon.com/`
[3] `http://www.fico.com/`
[4] `http://www.forrester.com/go?docid=39088`

Haley Limited

Haley Limited se forma en 2007 cuando la empresa *Rule Burst* es adquirida por *Haley Systems*[5]. Ofrece una suite compuesta por los siguientes productos:

- *Haley Office* y *Web Rules*: para administrar las reglas de negocio.
- *Haley Expert Rules*: para mantener versionado y autoría de reglas de negocio.
- *Haley Business Rules Engine*: para ejecutar las reglas en el motor de ejecución.

En Octubre de 2008 es adquirida por *Oracle Corporation* [65].

IBM ILOG

IBM ILOG[6] es una de las denominadas empresas "enterprise-class" dentro de los motores de evaluación de reglas de negocio.

Posee la mayor base instalada y la mayor penetración en el mercado según análisis de las consultoras IDC, Forrester y Gartner. Su solución combina un motor de reglas para diferentes plataformas (*JRules* para *Java* , *C++*, *Rules for .Net* para la plataforma .Net y sistemas legados como por ejemplo sistemas Cobol en Mainframe).

En Julio de 2008 *IBM* anuncia la adquisición de ILOG [30] convitiéndose en *IBM ILOG*.

Innovations Software

Innovations Software es una empresa basada en Alemania, una de las líderes para los desarrolladores *Java* que quieran utilizar un motor de evaluación de reglas. Provee un mecanismo interesante para la creación de reglas utilizando un "flujo visual"; su producto relacionado al área es *Visual Rules*.

JBoss

JBoss forma parte de *Red Hat*, uno de los líderes en el desarrollo de Middleware. Dentro de lo que es *JBoss Jems* se ofrece a *JBoss Rules* y *Drools* versión paga y open-source respectivamente de un motor de ejecución de reglas para la plataforma *Java*.

[5] `http://www.haley.com/`
[6] `http://www-01.ibm.com/software/websphere/ilog_migration.html`

Pegasystems

Con la herramienta *PegaRULES*, la empresa *Pegasystems*[7] con base en Cambridge USA, es una de las empresas más fuertes dentro del mundo de reglas de negocio en conjunto con BPM [139].

Posee (según análisis de la empresa Forrester) más de 200 clientes con más de 300 instalaciones que utilizan todas las versiones de su herramienta.

RulesPower

RulesPower es una empresa co-fundada en 2001 por Charles Forgy inventor de los algoritmos *Rete*, *ReteII* y *ReteIII*.

En Setiembre de 2005 [19] la empresa *RulesPower* fue adquirida por *FICO* obteniendo la licencia para integrar *ReteIII* en su herramienta *Blaze Advisor*.

SAP

SAP[8] es una empresa proveedora de servicios y aplicaciones. En Octubre de 2007 la empresa adquiere tecnología de reglas de negocio a través de la absorción de la empresa *Yasu Technologies* [91]. Dentro del roadmap de *SAP* se encuentra la integración del producto adquirido (*QuickRules*) como add-in dentro de su producto *NetWeaver*.

Yasu

Yasu Technologies[9] es una empresa basada en India que desarrolló *QuickRules*, adquirida por *SAP* a finales de 2007 [91].

InRule Technology

InRule Technology[10] es una empresa de desarrollo de software con base en Chicago especializada en el desarrollo de un motor de evaluación de reglas (*InRule*) para la Plataforma .Net.

Sandia National Laboratories

Sandia National Laboratories[11] es un organismo gubernamental de los Estados Unidos, creador del producto *Jess* específico para la plataforma Java.

[7] `http://www.pegasystems.com/`
[8] `http://www.sap.com/index.epx`
[9] `http://www.yasutech.com/`
[10] `http://www.inrule.com/`
[11] `http://www.sandia.gov/`

Ness Technologies

Ness Technologies[12] es una empresa proveedora de servicios de Tecnologías de Información creadora de la herramienta *USoft Rules*.

Delta-R

Delta-R[13] es una compañía española especializada en la gestión avanzada de reglas de negocio; desarrolla la plataforma *OnRules*.

Computer Associates

Fundada en 1976, *CA - Computer Associates*[14] es una compañía global que tiene su casa matriz en Islandia, NY, operando en más de 100 países con más de 150 oficinas.

Relacionado con los motores de evaluación de reglas de negocio, compite con el producto *CA Aion Business Rules Expert*.

Versata

Versata Logic Suite[15] es una de las primeras empresas en ofrecer una suite de productos relacionados con reglas de negocio. La empresa fue adquirida por *Trilogy*[16] en el 2005 [102].

B.2. Herramientas

Existe una cantidad importante de herramientas en el mercado de reglas de negocio tanto de código abierto como comerciales.

La Figura B.1 detalla la evolución de las herramientas relacionadas con la administración de reglas de negocio desde sus orígenes hasta llegar a la actualidad [128]. Resulta interesante destacar que en dicho gráfico se explicita como ha sido la adquisición de herramientas en el mercado desde el inicio de la década del 80, cuando se produjo la revolución de los Sistemas Expertos.

[12] `http://www.ness.com/GlobalNess`
[13] `http://www.delta-r.com/`
[14] `http://www.ca.com/`
[15] `http://www.versata.com/`
[16] `http://www.trilogy.com/`

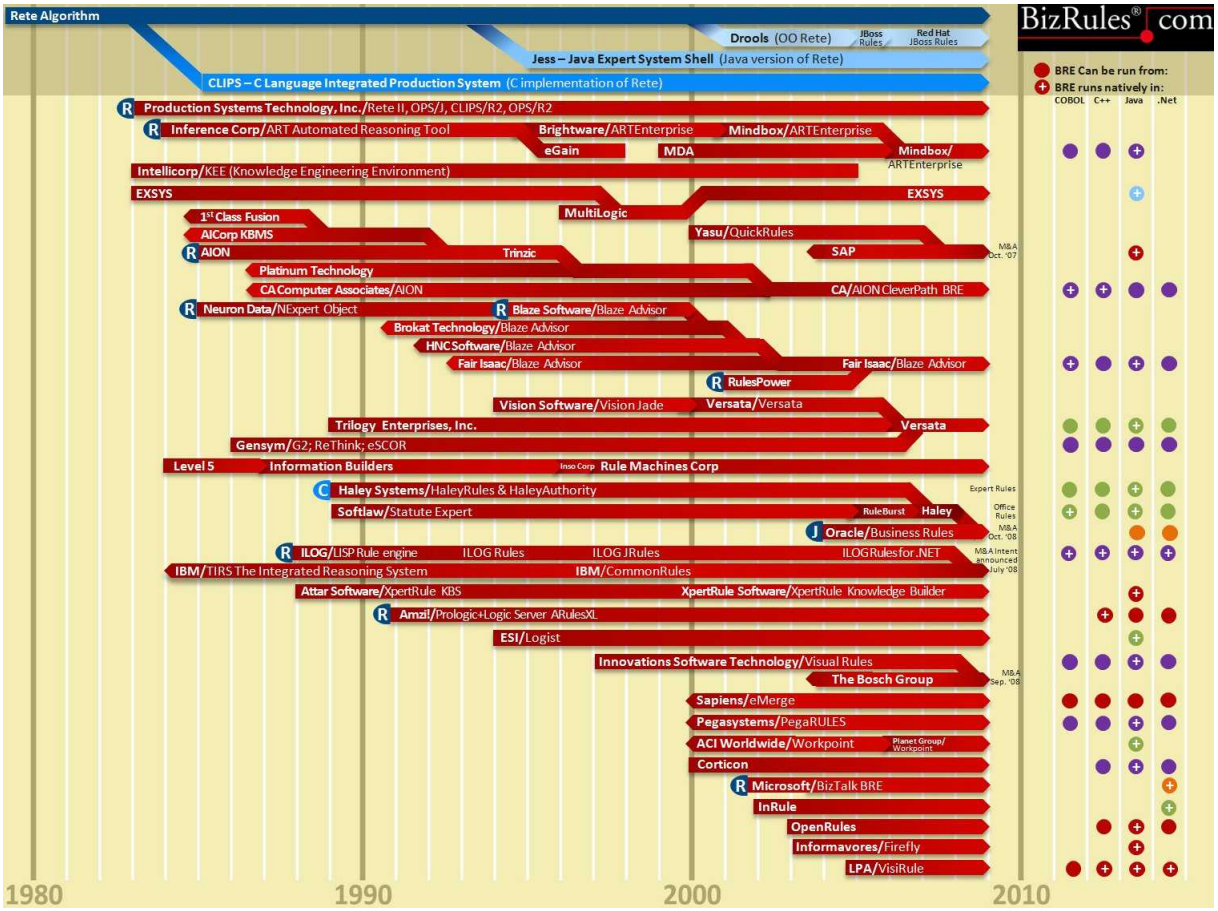


Figura B.1.: Familia de herramientas

Para realizar el análisis de herramientas se categorizan en dos grandes grupos: de código abierto (open-source) y comerciales o propietarias.

Herramientas Open Source

Detalle de herramientas de código abierto para *Java*[17]:

- Drools.
- SweetRules.
- Mandarax.
- JLisa.
- OpenRules.

[17] `http://java-source.net/open-source/rule-engines`

B. Empresas y Herramientas

- Zilonis.
- Hammurapi Rules.
- Take.
- OpenLexicon.
- JEOPS.
- JRuleEngine.
- Prova.
- CommonRules.
- Jena.

Herramientas de código abierto para la plataforma *.Net*[18].

- NxBRE.
- Drools.NET.
- Simple Rule Engine.

Para el lenguaje *Python*:

- Pyrete.
- Pychinko.
- PyCLIPS.

Para el lenguaje *Ruby*:

- Rools.
- Ruleby.
- SIE.
- Khammurabi.

Herramientas específicas al lenguaje de programación *C*:

- Clips.

Herramientas desarrolladas en *Javascript*:

- JSONrules.

[18] `http://csharp-source.net/open-source/rule-engines`

JxBRE

JxBRE [48] es un motor de evaluación de reglas de negocio open-source implementado en *Java*. El proyecto fue discontinuado en el 2004, fue superado por el proyecto *Drools*.

Jena

Jena [40] es un framework open source (licencia BSD[19]) para la Web Semántica, originario de HP Labs[20]. Si bien incluye un motor de reglas genérico, está enfocado fuertemente dentro del marco de la Web Semántica, proveyendo un ambiente programático para interactuar con RDF, RDFS, OWL y SPARQL [96].

Incluye los siguientes razonadores[21]:

- *Transitivo*: implementa las propiedades simétrica y transitiva de `rdfs:subPropertyOf` y `rdfs:subClassOf`, se utiliza como bloque constructivo de razonadores más complejos.
- *RDFS*: implementa un subconjunto configurable de la especificación de links RDFS según se describe en *RDF Semantics*[22]. Existen varios niveles de conformidad:
  - *Full*: implementa todos los axiomas RDFS y reglas de clausura exceptuando el nodo *bNode*. Es el modo más expresivo y más costoso ya que todas las sentencias deben ser chequeadas.
  - *Default*: omite los chequeos que se realizan en el modo anterior, incluye reglas axiomáticas.
  - *Simple*: clausura transitiva de las relaciones `rdfs:subPropertyOf` y `rdfs:subClassOf`; omitiendo todos los axiomas.
- *OWL*, *OWL Mini*, *OWL Micro*: un subconjunto incompleto de *OWL/Lite*, parte del lenguaje *OWL/Full*.
- *DAML micro*: utilizado internamente para habilitar la API legada DAML, para proveer inferencia básica sobre RDFS.
- *Genérico (de propósito general)*: un razonador basado en reglas que soporta razonamiento hacia adelante, hacia atrás y una estrategia híbrida sobre *grafos RDF*. Existen dos motores de reglas internos; denominados *forward chaining RETE engine* basado en el algoritmo *Rete* [150] y *tabled datalog engine* pudiéndose ejecutar en forma separada. El razonador que ejecuta en modo

[19] `http://www.opensource.org/licenses/bsd-license.php`
[20] `http://www.hpl.hp.com/semweb/`
[21] `http://jena.sourceforge.net/inference/index.html`
[22] `http://www.w3.org/TR/rdf-mt/`

hacia atrás utiliza un motor de programación lógica con una estrategia de ejecución similar a *Prolog*, donde las reglas se ejecutan de arriba hacia abajo, izquierda a derecha utilizando backtracking (SLD resolution).

Drools

Drools [13] es un motor de inferencia basado en reglas de tipo inferencia hacia adelante para Java.

Fue adquirido por *JBoss* y pasó a llamarse *JBoss Rules*[23] incluido dentro de lo que se denomina JEMS (JBoss Enterprise Middleware Systems[24]); se clasifica dentro de los motores de ejecución de reglas como Production Rule System (Turing Complete).

Provee un lenguaje declarativo lo suficientemente flexible para especificar la semántica de cualquier dominio con varios lenguajes específicos de dominio.

Características:

- Utiliza lenguajes específicos de dominio para que tanto los analistas de negocio como desarrolladores puedan detallar las reglas en forma natural y provee deferentes interfaces para su definición.
- Administra las reglas de negocio en forma independiente del código de la aplicación.
- Incluye un complemento para usarse de forma integrada a Eclipse.
- Incluye un *BRMS* basado en Web para centralizar y administrar las reglas de negocio.

SweetRules

SweetRules (Semantic WEb Enabling Technology) [97] posee un conjunto integrado de herramientas para ontologías y reglas en Web Semántica utilizando el lenguaje *RuleML* o el estándar *SWRL*. Soporta el manejo de inferencias tanto hacia atrás como hacia adelante.

OPSJ

OPSJ [64] es una herramienta desarrollada en *Java* que agrega la posibilidad de ejecutar reglas de negocio. Utiliza el algoritmo *ReteII*, soportando razonamiento hacia adelante y hacia atrás.

Es una de las herramientas pioneras en lo que respecta a la investigación del procesamiento de reglas en forma paralela [166], es decir explotar el paralelismo que se puede lograr al ejecutarse en múltiples procesadores o núcleos (cores).

[23] `http://www.jboss.com/products/platforms/brms/`
[24] `http://www.jboss.com/products/index`

Mandarax

Mandarax [51] es una implementación open source en *Java* de un motor de reglas. Soporta varios tipos de hechos y reglas basados en reflection, bases de datos, EJB así como también estándares como *RuleML*.

Principales características:

- Basado en razonamiento hacia atrás.
- Integrable a todo tipo de orígenes de datos (bases de datos, Web Services, EJBs).
- Provee un motor de inferencia compatible con *J2EE*.
- Permite persistir bases de conocimiento en un formato similar a *RuleML*.

El proyecto ya no se encuentra activo, existen dos derivados de ésta herramienta:

- *Take*.
- *Prova*.

JLisa

JLisa [45] es un framework para construir reglas de negocio compatible con el estándar *JSR-94* [36]. La última versión del proyecto fue actualizada el 20/11/2003.

Open Rules

Open Rules [62] es un framework open source de administración de reglas de negocio para *Java*. Características principales:

- Se puede utilizar en cualquier ambiente J2EE/J2SE.
- Utiliza *Excel* o *Google Spreadsheets* como editor de las reglas.
- Tiene un plugin en Eclipse para integrarse al ambiente de desarrollo *Java*.
- Soporta la externalización de las reglas por medio de tablas de decisión.
- Soporta la especificación *JSR-94* además de tener una API propia.
- No tiene capacidad de inferencia, simplemente ejecuta el conjunto de reglas que se encuentre definido en las planillas Excel a través de las tablas de decisión.

Zilonis

Zilonis [111] es una plataforma de reglas de negocio y ambiente de ejecución de scripts. El núcleo de la herramienta está basado en una variación del algoritmo *Rete*.

Hammurapi Rules

Hammurapi Rules [29] es un motor de evaluación de reglas de tipo forward chaining compatible con el estándar *JSR-94*.

Las reglas se definen directamente utilizando el lenguaje *Java*. No hay pasos de interpretación, una vez que el conjunto de reglas se instancia a partir de definiciones en XML las mismas se compilan a bytecode *Java*.

Take

Take [99] es un lenguaje de scripting utilizado para definir reglas de derivación, donde un compilador crea código *Java* a ser utilizado por la aplicación; inspirado por *Mandarax*.

Open Lexicon

Open Lexicon [61] es un proyecto open source que integra un motor de evaluación de reglas de negocio y un administrador de procesos.

Incluye un repositorio de reglas de negocio, motor de ejecución e interfaces basadas en Web para administrar y testear las reglas. También incluye herramientas de administración de procesos para orquestar interacciones entre objetos y reglas de negocios.

JEOPS

JEOPS (Java Embedded Object Production System) [43] es un motor de evaluación de reglas de tipo inferencia hacia adelante; no registra movimiento desde el 28/09/2003.

JRule Engine

JRule Engine [46] es un motor de evaluación de reglas de negocio basado en el estándar *JSR-94* que soporta razonamiento hacia adelante. La última actualización del proyecto es del 16/04/2008.

Prova

Prova (Prolog+ Java) [70] es un sistema basado en reglas que extiende el motor *Mandarax*, combinando los beneficios de la programación declarativa y la orientación a objetos. Combina en su lenguaje la programación imperativa y declarativa.

Se posiciona como un sistema de reglas basado en scripting para utilizarse como middleware.

CommonRules

CommonRules [7] es un framework para el desarrollo basado en reglas con énfasis en la separación de la lógica de negocio de los datos, desarrollado para *Java*. Fue un proyecto de investigación de IBM que fue cancelado.

Entre otras cosas definió un lenguaje de definición de reglas de negocio denominado *BRML* que evolucionó en *RuleML* [86].

NxBRE

NxBRE [55] es un motor de evaluación de reglas de negocio open source implementado en *.NET* basado en *JxBRE*.

Ofrece dos aproximaciones diferentes para utilizarlo: como motor de flujo o como motor de inferencias. La primera opción utiliza un archivo XML como forma de controlar el flujo de proceso; este archivo contiene instrucciones de tres tipos: reglas, evaluaciones lógicas y estructura. La segunda opción utiliza un motor de inferencia basado en el razonamiento hacia adelante soportando conceptos de *RuleML*.

La definición de reglas se puede utilizar con la notación *RuleML* (atoms, facts, queries, implications, slots) o un formato propietario. La última actualización del proyecto es del 06/01/09; versión 3.2.0.

Drools.NET

Drools.NET [14] es un port para .Net del motor de evaluación de reglas *Drools*. La última actualización del proyecto data del 27/02/07, soportando la mayoría de las funcionalidades de *Drools 3.0*.

Simple Rule Engine

SRE (Simple Rule Engine) [95] es un motor de evaluación de reglas de negocio de razonamiento hacia adelante para la plataforma .Net. La última actualización del proyecto se registra el 16/06/2006; versión 2.2.

Expergent

Expergent [16] es un motor de evaluación de reglas desarrollado en .Net. Características principales:

- Implementa el algoritmo *Rete*.
- Define un lenguaje propietario *BOO* para la creación y compilación de reglas.
- Implementa la resolución de conflictos.

La última actualización del proyecto se registra el 16/07/2008.

Pyrete

PyRete [74] es un motor de evaluación de reglas implementado en *Python* diseñado como un clon de *Clips*.

Pychinko

Pychinko [72] es una implementación del algoritmo *Rete* en *Python* desarrollado en la Universidad de Maryland.

PyCLIPS

PyCLIPS [73] es un motor de evaluación de reglas para *Pyton* que embebe a *Clips*.

Clips

CLIPS [11] es una herramienta de dominio público utilizado para construir sistemas expertos. El nombre es un acrónimo de "C Language Integrated Production System".

La sintaxis y nombre fue inspirado por Charles Forgy. La primer versión fue desarrollada en 1985 en la NASA como una alternativa a un sistema experto ya existente denominado ART*Inference. Incorpora un lenguaje orientado a objetos denominado *COOL*.

Existen varios descendientes del lenguaje Clips como por ejemplo Jess [44], FuzzyClips [23] y EHSIS[25].

FuzzyClips

FuzzyCLIPS [23] es una extensión de *CLIPS* que incorpora la capacidad de trabajar con lógica difusa [211].

Rools

Rools [84] es un motor de reglas desarrollado en *Ruby*. Se permiten definir entre otras cosas tablas de decisión en formato XML o como clases *Ruby*.

Ruleby

Ruleby [85] es un motor de reglas desarrollado en *Ruby*, es probablemente uno de los más populares y maduros en el mundo *Ruby*.

Provee un DSL [169] para definir e insertar las reglas (de tipo regla de producción) en la memoria de trabajo del motor de evaluación; los hechos son los propios objetos Ruby. El motor de inferencia utiliza el algoritmo Rete totalmente implementado en Ruby.

[25] `http://erabaki.ehu.es/ehsis/`

SIE

SIE [94] es un motor de inferencia desarrollado en *Ruby*. No utiliza el algoritmo *Rete*, está basado en *ESIE*.

JSONrules

JSONrules [47] es un motor de evaluación de reglas desarrollado en *Javascript*. Características principales:

- Utiliza *DOM* para poder referenciar elementos del *HTML*.
- La memoria de trabajo es el propio documento *DOM*.
- Implementa razonamiento hacia delante.
- Utiliza *RIF* y *R2ML*.

Khammurabi

Khammurabi [49] es un motor de reglas desarrollado en *Ruby*, implementa el algoritmo *Rete*. La última actualización del proyecto data del 04/08/2005.

Herramientas Comerciales

Las principales herramientas comerciales se detallan a continuación.

- *WebSphere ILOG JRules* y *Rules for .Net* de *IBM ILOG*.
- *Jess* de *Sandia National Laboratories*.
- *Blaze Advisor* de *FICO*.
- *MSBRE* y *Windows Workflow Foundation Rules Engine* de *Microsoft*.
- *InRule* de *InRule Technology*.
- *Corticon Studio* de *Corticon Technologies*.
- *Haley Business Rules Engine* de *Haley Limited*.
- *CleverPath Aion Business Rules Expert* de *Computer Associates*.
- *Oracle Rules Engine* de *Oracle*.
- *PegaRULES* de *Pegasystems*.
- *USoft Rules* de *Ness Technologies*.

- *QuickRules* de *Yasu Technologies*.
- *SAP Netweaver BRM* de *SAP*.
- *Visual Rules* de *Innovations Software*.
- *OnRules* de *Delta-R*.

Jess

Jess [44] es un motor de reglas comercial desarrollado en los laboratorios *Sandia National Laboratories*. Características principales:

- Las reglas se escriben utilizando notación del tipo Lisp [113] o Prolog [216].
- Inicialmente derivó de *CLIPS* y se reescribe en *Java*.
- Utiliza un lenguaje propio de definición de reglas denominado *jessML*.
- Las últimas versiones incluyen un plugin para Eclipse.
- Utiliza una versión mejorada del algoritmo *Rete*.

Visual Rules

Visual Rules [105] es un motor de evaluación de reglas para *Java* desarrollado por *Innovations Software* con fuerte enfoque de orientación a servicios en donde las reglas se generan en código *Java* pudiéndose exponer como *Web Services*.

Se compone de:

- *Visual Rules Modeller*: editor gráfico de reglas integrado totalmente a Eclipse en donde se define el modelo de reglas de negocio, documentación y se genera el código *Java* asociado.
- *Visual Rules Database Connectivity*: permite acceder en forma eficiente a diferentes bases de datos relacionales durante la ejecución de reglas.
- *Visual Rules Rule Server*: servidor en donde se centraliza la administración de reglas.
  - Permite realizar el despliegue (deploy) de las reglas sin interrumpir la ejecución de las mismas.
  - Dispone de una consola Web para monitorear el motor de ejecución.
  - Ejecuta en cualquier servidor de aplicaciones JavaEE.
- *Visual Rules Build Tools*: automatiza la generación de código, ejecución de reglas, testing y análisis de cobertura de código.

QuickRules

QuickRules es un motor de evaluación de reglas de negocio para *Java* y *.Net* desarrollado por la empresa *Yasu Technologies* adquirido en el 2007 por *SAP* [91]. Principales características:

- Versionado de reglas que permite volver atrás cualquier tipo de cambios.
- Permite definir reglas tanto en lenguaje natural como en un lenguaje propietario.
- Detección de conflictos entre reglas.
- Permite cambios en reglas y términos en tiempo de ejecución.
- Posee un motor de workflow incluido.
- Permite especificar el orden de ejecución de las reglas por medio de diagramas.

SAP Netweaver Business Rules Management

SAP Netweaver BRM [90] es la plataforma que fue adquirida a *Yasu Technologies* integrada al producto *NetWeaver*.

Compuesto por las siguientes herramientas:

- *Compositor*: para la creación y modificación de reglas de negocio utilizando diferentes formatos como por ejemplo tablas de decisión.
- *Analizador*: permite que los expertos del negocio realicen tests, refinen, analicen y optimicen las reglas de negocio.
- *Administrador*: Interfaz Web para administrar las reglas de negocio.
- *Repositorio*: Provee un ambienta para el manejo de versiones, permisos, control de acceso, alertas relacionado con las reglas de negocio.
- *Motor de ejecución*: ejecuta las reglas, integrado a la tecnología provista por *SAP NetWeaver Composition Environment*.

Rule Burst

Rule Burst es un BRMS de la empresa del mismo nombre con base en Canberra, Australia. La empresa fue adquirida por *Haley Limited* en Noviembre de 2007 [20].

USoftRules

USoft Rules [104] es un motor de evaluación de reglas desarrollado por la empresa *Ness Technologies*; en donde se crea un esquema de objetos sobre la base de datos. Las reglas de negocio se escriben en SQL estándar y se relacionan al núcleo de la aplicación.

PegaRULES

PegaRules [87] es un motor de evaluación de reglas que separa la lógica del negocio de las aplicaciones de misión crítica permitiendo que el negocio capture, administre y ejecute las prácticas y políticas de negocio definidas.

Características principales:

- Manejo de diferentes tipos de reglas para facilitar la definición y uso.
  - *Declarativas*: reglas que realizan cálculos o verifican que se cumplan las restricciones.
  - *Árbol de decisión*: definen inferencias basadas en hechos para ejecutar lógica del tipo: "*If - then*".
  - *Integración*: reglas para facilitar la interfaz entre sistemas heterogéneos.
  - *Procesos*: reglas que administran el recibo, asignación, ruteo y seguimiento del trabajo a realizarse.
- Soporte de inferencias hacia atrás y hacia adelante.
- Administración centralizada de las reglas, permitiendo una ejecución distribuida de las mismas.
  - Soporte de diferentes sistemas operativos estándar: Windows, Solaris, z/OS, AIX, Linux.
  - Soporte de los servidores de aplicaciones líderes incluyendo IBM WebSphere, BEA WebLogic, Apache Tomcat.
  - Procesos y reglas de negocio residen en un servidor de bases de datos centralizado: Oracle, Microsoft SQL, y/o IBM DB2.
- Optimizado para ejecutar en ambientes Java.
- Construcción de reglas en forma gráfica utilizando *Microsoft Visio*.

Oracle Rules Engine

Oracle Rules Engine [66] está conformado por un motor de inferencia, un lenguaje de reglas propietario denominado *Rule Language* y una herramienta para la edición de reglas.

- *Rule Author*: se utiliza mediante un browser para crear y editar reglas de negocio.
- *Rule language* (RL): es el lenguaje de programación de reglas integrado a Java.
- *Motor de ejecución*: utiliza un derivado de *Jess*.

- *Rule SDK*: consta de una API para que el desarrollador pueda definir objetos del negocio, ejecutar y monitorear las reglas.
- Compatible con el estándar *JSR-94*.

CleverPath Aion Business Rules Expert

CleverPath Aion Business Rules Expert [10] es la solución de *Computer Associates* con un fuerte enfoque en el desarrollo basado en reglas. Consta de un motor de inferencia para ejecutar reglas en tiempo real con razonamiento hacia atrás y adelante.

Soporta plataforma windows, z/OS, Unix, en particular aplicaciones basadas en mainframe.

Haley Business Rules Engine

Haley Business Rules Engine [28] es un motor de evaluación de reglas previamente denominada *HaleyRules* desarrollado por *Haley Limited*.

Utiliza una versión optimizada del algoritmo Rete funcionando en modo razonamiento hacia adelante y hacia atrás.

Ofrece un soporte multiplataforma para *Java*, *.Net*, *C/C++*, *RPG IV (ILE RPG)* pudiendo ser embebido en las aplicaciones o ser configurado para utilizarse mediante Web Services.

Corticon Studio

Corticon Studio [12] es la solución de administración de reglas de negocio de la empresa *Corticon Technologies*. Ofrece una infraestructura que permite especificar las reglas de negocio en lenguaje natural en un modelo centralizado que posteriormente pueden ser ejecutadas en ambientes heterogéneos como .Net, j2EE, servicio de decisión utilizando un esquema de orientación a servicios o integrado a un administrador de procesos de negocio.

La suite consta de los siguientes productos:

- *Corticon Business Rules Modeling Studio*: herramienta standalone utilizada para modelar, analizar, testear las reglas de negocio. Se comunica en forma automática con el repositorio central de reglas para compartir vocabulario, juego de reglas, escenarios de test, etc.
- *Corticon Business Rules Server*: es el motor de ejecución en sí.
- *Corticon Enterprise Data Connector*: Integra los modelos de decisión con datos externos, es decir permite conectarse con fuentes heterogéneas, entre ellas diferentes bases de datos, data warehouses, documentos, colas de mensajes etc.

InRule

InRule [32] es un motor de reglas de negocio que extiende *Windows Workflow Foundation* desarrollado por *InRule Technology* específico para la Plataforma *.Net*. Provee tecnología de reglas de negocio para la autoría, verificación y administración centralizada de reglas.

Principales características:

- Facilidad para la creación de reglas, mediante herramientas que permiten a los desarrolladores y analistas de negocio crear, testear y mantener reglas.
- Inspección en tiempo real del resultado de la ejecución de reglas.
- Creación de la interfaz de usuario en forma dinámica basado en las reglas.
- Interoperable vía Web Services cumpliendo con la especificación *WSE 2.0*.
- Uso de tablas de decisión y más de 90 funciones "built-in".
- Catálogo de reglas para administrar operaciones como check-in\check-out, versionado y permisos.

Componentes principales:

- *irAuthor*: herramienta principal para la autoría y mantenimiento de reglas.
- *irVerify*: herramienta de test de reglas interactiva; utilizada para ejecutar y realizar debug de aplicaciones que utilicen reglas de negocio permitiendo la creación de casos de test para ser reutilizados.
- *Microsoft Windows Workflow Foundation Rules Engine*: Parte de la plataforma *.Net 3.0*; en particular del componente *Windows Workflow Foundation* utilizado para la ejecución de las reglas.
- *irDeveloper*: integra *inRule* en las propias aplicaciones.
- *irCatalog*: catálogo para almacenar, versionar y administrar reglas.
- *RuleTime*: extiende el motor de reglas de WWF con las de 90 funciones built-in; permite que las reglas envíen notificaciones a los usuarios entre otras funcionalidades.

Microsoft Business Rules Engine

MsBRE [101] es un motor de ejecución de reglas de tipo razonamiento hacia adelante incluido en la distribución de *Biztalk Server*. Consta de los siguientes componentes:

- *Business Rules Composer*: para construir las políticas de negocio; provee interfaces para que los desarrolladores, administradores y analistas de negocio puedan:
  - Crear definiciones de vocabularios y relacionarlos con las fuentes de datos (elementos de XML Schema, bases de datos y clases .NET).
  - Incluir definiciones de vocabularios en reglas.
  - Componer versiones de políticas a partir de un grupo de reglas.
  - Publicar vocabularios y pasar a producción las políticas.
- *Rules Engine Deployment Wizard*: para poner en producción las políticas creadas.
  - Exportar e importar políticas desde o hacia XML o bases de datos.
  - Publicar o eliminar de producción una versión de una política determinada.
- *Run-time Rule engine*: ejecuta las políticas definidas.
  - Procesa las reglas creadas en forma declarativa.
  - Agrupa todas las reglas que aplican al mismo proceso de negocio.

El repositorio de reglas centraliza las políticas y vocabularios, se almacena en una base de datos SQL Server.

Blaze Advisor

Blaze Advisor [21] es una herramienta desarrollada por la empresa *FICO*. Ofrece un entorno para crear, probar y desarrollar reglas de negocio; es una de las herramientas más importantes del mercado.

Características:

- Administración del repositorio de reglas centralizado pudiéndose almacenar en archivos XML, bases de datos o directorios LDAP.
- Soporte de tablas y arboles de decisión.
- Seguimiento y versionado de reglas.

- Las reglas se escriben en una sintaxis similar al idioma inglés, por lo que los usuarios no técnicos pueden crearlas y mantenerlas. Internamente utiliza un lenguaje propio de definición de reglas denominado *Structured Rules Language*.
- La definición de reglas utilizan un vocabulario independiente de la plataforma denominado *Business Object Model Adapter*.
- Repositorio único para múltiples plataformas de ejecución nativas (Java, .NET, COBOL), pudiendo utilizarse también desde otros ambientes utilizando Web Services.
  - *Entorno Java*: es un producto totalmente *Java* que se ejecutará en cualquier plataforma que admita una Máquina Virtual Java.
  - *Entorno COBOL*: utiliza el entorno de desarrollo *Java* y genera *COBOL*.
  - *Entorno .NET*: es un producto para utilizarse en la plataforma *.NET*.
- Soporta múltiples modos de ejecución: inferencia (hacia adelante, hacia atrás), secuencial y secuencial compilado.
- Brinda un framework para test unitario de reglas de negocio denominado *brUnit*.
- Incorpora el algoritmo *ReteIII*, adquirido cuando se realizó la adquisición de la empresa *RulesPower* [19].

FICO también provee soluciones verticales basadas en reglas para:

- Análisis de préstamos (producto LOAN).
- Detección de fraudes (producto FALCON).

WebSphere ILOG JRules

WebSphere ILOG JRules [107] es un motor de evaluación de reglas de negocio para *Java*, es una de las herramientas más importantes del mercado.

Consta de los siguientes componentes:

- *Rule Builder*: Es un entorno gráfico para el desarrollo, test y gestión de las reglas de negocio. Permite definir reglas utilizando diferentes formatos, una sintaxis muy similar Java y también un lenguaje propietario de definición de reglas denominado BAL (Business Action Language).
- *JRules Repository*: Es el repositorio central de las reglas de negocio.
- *Business Rule Editor*: Editor gráfico para escribir reglas de manera sencilla e intuitiva.

- *JRules Rule Server*: motor de inferencia para ser utilizando en un entorno J2EE\J2SE [35, 37]. El motor de JRules consta de un conjunto de clases Java con la lógica necesaria para ejecutar reglas. Representa a los objetos de la aplicación directamente con código Java sin la necesidad de interpretar código en tiempo de ejecución.

Rules for .Net

Rules for .Net [108] es un motor de evaluación de reglas de negocio para la plataforma *.Net 3.0* integrado con Microsoft Windows Workflow Foundation desarrollado por la empresa IBM ILOG.

Se dispone de un repositorio accesible via Sharepoint Services, un conjunto de plug-ins para Visual Studio y Word.

Los programadores se encargan de crear las reglas en Visual Studio, las exportan como documentos de tipo RuleDocs, un formato basado en XML y posteriormente los analistas de negocio las editan utilizando Word. También se dispone de complementos para Microsoft Excel para permitir crear reglas para BizTalk.

OnRules

onRules [59] es un BRMS que permite a los usuarios definir y gestionar todo tipo de modelos y reglas de negocio de forma ágil y dinámica.

Está construido sobre la plataforma empresarial de Java (J2EE), 100 % orientado a servicios cumpliendo con el estándar SOAP.

Algunas funcionalidades que provee:

- Modelos de scoring.
- Redes neuronales.
- Arboles de decisión.
- Flujo de reglas.

WWF

Microsoft Windows Workflow Foundation (WWF) [109] forma parte de la plataforma *.Net 3.0*.

Es el modelo de programación, motor y herramientas para la construcción de aplicaciones que utilicen workflow en ambientes Windows.

Provee un motor de workflow in-process, un diseñador integrado a Visual Studio y un motor de evaluación de reglas que permite definir reglas en forma declarativa,

realizar desarrollos de workflows basado en reglas integrable a cualquier aplicación .Net denominado Windows Workflow Foundation Rules Engine.

Las reglas se exponen de dos maneras, como condiciones en actividades (*Activity Conditions*) o como juegos de regla dentro de actividades de tipo Política (*Policy activities*).

Las *Activity Conditions* se utilizan para influenciar en el comportamiento de la ejecución de actividades, determinando que código debe ejecutarse. Las mismas pueden especificarse como *CodeConditions* indicando que se debe declarar un manejador de dicho código o como *RuleConditionReference* en donde las condiciones se separan en archivos de reglas en el contexto del proyecto de workflow. En este caso el beneficio de *RuleConditions* vs *CodeConditions* es claramente que las reglas si bien son parte del modelo, pueden ser actualizadas dinámicamente durante la ejecución de las instancias de proceso.

C. Estándares y Lenguajes de Reglas

Existe una cantidad interesante de lenguajes y estándares relacionados al concepto de reglas de negocio. A nivel académico podemos ver que existen dos grandes enfoquen llevados adelante por lo que es la W3C [110] y OMG [57] que la industria no está siguiendo, ya que la mayoría de las herramientas del mercado utilizan variaciones de lo que son las *reglas de producción* representadas por el estándar *PRR* [71].

Para facilitar el análisis se categorizan a los lenguajes y estándares en dos grandes grupos: independientes de la plataforma y propietarios.

C.1. Independientes de la Plataforma

Detalle de lenguajes independientes de la plataforma:

- CommonRules.
- SWRL.
- RuleML.
- URML.
- R2ML.
- SBVR.
- RIF.
- PRR.
- OCL.
- ODM.
- OWL.
- OWL 2.

CommonRules

Proyecto de investigación de IBM que definió entre otras cosas un lenguaje de definición de reglas de negocio denominado *BRML* que evolucionó en lo que actualmente se conoce como *RuleML*.

SWRL

Semantic Web Rule Language (SWRL), subconjunto expresivo de las reglas que se pueden expresar en *RuleML*.

RuleML

Rule Markup Language [86]; es un lenguaje basado en XML para especificar reglas.

Su objetivo es crear un conjunto de lenguajes de marcado neutral para lograr interoperabilidad semántica entre diferentes motores de reglas de negocio. Para ello, sus fundadores han elegido enfocarse en intercambios entre estándares de la industria en vez de prototipos académicos.

Permite entre otras características el almacenamiento, intercambio, búsqueda y activación de reglas. Sus primeras versiones estaban basadas únicamente en XML [18], pero en las siguientes versiones se combina XML con RDF.

Es un estándar que consta de una gran gama de sublenguajes que están evolucionando permanentemente. Su extensibilidad permite que nuevos sublenguajes se puedan agregar con facilidad. Entre sus lenguajes se destacan una implementación de Datalog (un lenguaje lógico desarrollado para el modelo relacional cuyo poder de expresividad equivale al álgebra relacional con el agregado de la recursión); una implementación de lógica de primer orden (First Order Logic) y una implementación denominada OO RuleML, orientada a objetos.

Dada la importancia que se le da a la interacción entre sistemas, se están desarrollando traductores entre diferentes motores de reglas de negocio y RuleML.

Se detalla en la Figura C.1 los diferentes sublenguajes que provee *RuleML*.

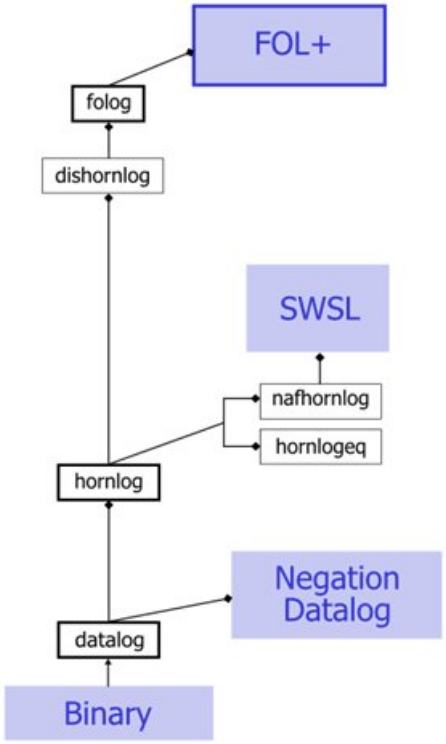


Figura C.1.: Sublenguajes en RuleML

URML

URML (UML-Based Rule Modeling Language) [176]; lenguaje basado en UML muy cercano a R2ML para la representación de reglas de negocio desarrollado por el grupo REWERSE. Su objetivo principal es proveer una forma constructiva visual de reglas tomando como referencia el modelo UML. Soporta regla de derivación, producción y reacción.

R2ML

REWERSE Rule Markup Language (R2ML) es desarrollado por el grupo REWERSE con el objetivo de tener un mecanismo de intercambio de reglas entre diferentes herramientas.

Características:

- Lenguaje de reglas basado en XML.
- Soporta la definición de reglas de integridad, derivación, producción y reacción.
- Modelado utilizando el enfoque MDA, los conceptos relacionados con reglas se definen con la ayuda de MOF y UML.

SBVR

Semantics and Vocabulary of Business Rules (Semántica de Vocabulario y Reglas de Negocios) [93] está formalmente definido como una taxonomía que describe operaciones y reglas de negocio elementales. Es una especificación propuesta inicialmente por la *Business Rule Group* [100] y tomada por la OMG [57] en su meta modelo expresado en UML, integrado dentro de lo que se denomina Model-Driven Architecture (MDA) [191].

Detalla cómo las reglas y vocabulario de negocio se pueden capturar y especificar en términos no ambiguos en un lenguaje lógico formal con expresiones formuladas en un lenguaje natural restringido.

Por vocabulario nos referimos a todas aquellas palabras y términos que representan conceptos y operaciones en una organización.

Se enfoca principalmente en los expertos de negocios, no tiene en cuenta el procesamiento automático de las reglas; es decir, es para que los analistas de negocio puedan modelar el propio negocio, usando sus propios términos, independiente del manejo implícito o explícito que se haga desde los *sistemas de información*.

¿Qué tipos de reglas se pueden capturar? Solamente reglas en forma declarativa como el siguiente ejemplo:

> *It is permitted that* a rental is open *only if an* estimated rental charge is provisionally charged to *the* credit card of *the* renter of *the* rental.

Donde:

- Texto en itálica: especifica la sintaxis SBVR.
- Texto subrayado: referencia al vocabulario, es decir conceptos y términos de negocio.

El estándar SBVR, no incluye soporte de otro tipo de regla, por ejemplo del tipo *If – Then*, regla de flujos o tabla de decisión.

RIF

Rule Interchange Format [82] es un lenguaje basado en XML de la W3C para el intercambio de reglas entre ambientes de ejecución, por un lado entre reglas lógicas y de producción y también para interoperar con tecnologías relacionadas con Web semántica utilizando RDF, RDF Schema y OWL. La especificación alcanzó el estatus de candidato a recomendación en octubre de 2009.

Tiene como objetivo lograr establecer un lenguaje común (interlingua) para poder mapear diferentes lenguajes de reglas, permitiendo que las reglas específicas para un producto y plataforma se puedan compartir y reutilizar en otros motores de reglas.

RIF está diseñado como una familia de dialectos (ver Figura C.2) en donde cada uno es una colección de componentes que trabajan como un todo, formando una interlingua.

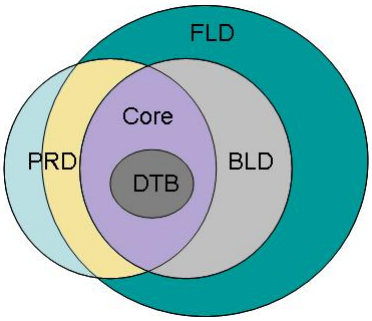


Figura C.2.: Familia de dialectos RIF

Los dialectos son: Basic Logic Dialect un formato de reglas lógicas, Production Rules Dialect[1] para representar reglas de producción y Core[2] detallando el conjunto común de los dos dialectos anteriores.

- RIF-FLD (RIF Framework for Logic-based Dialects) describe mecanismos para especificar sintaxis y semántica de dialectos RIF lógicos, actualmente existen dos dialectos:
  - RIF-BLD (Basic Logic Dialect o Dialecto Lógico Básico) es una especialización de RIF-FLD con capacidades de representación cláusulas de Horn con un numero de extensiones para soportar más expresividad como objetos y frames, IRIs (Internationalized Resource Identifiers) y tipos de datos de XML Schema.
  - RIF-PRD (Production Rules Dialect) es una especificación del dialecto de reglas de producción. El lenguaje de condiciones que se utiliza es un subconjunto de RIF BLD y RIF PRD.
- RIF-Core (Core Dialect) especifica un conjunto común utilizado por RIF-BLD y RIF-PRD incluyendo RIF-DTB.

PRR

Production Rule Representation [71] es un estándar de la OMG para proveer un meta modelo independiente de la plataforma (PIM – Platform Independent Model); define un modelo formal para las reglas de producción.

Se compone de:

[1] `http://www.w3.org/2005/rules/wiki/PRD`
[2] `http://www.w3.org/2005/rules/wiki/Core`

- Una estructura núcleo referida como *PRR Core* en donde se definen las reglas de producción para utilizarse en inferencia hacia adelante o procesamiento secuencial.

- Una extensión opcional que permite utilizar OCL[3] para definir el contenido semántico de las reglas.

Se prevee la existencia de un mecanismo de correlación con el estándar RIF para permitir intercambiar definiciones de reglas ejecutables entre modelos.

OCL

Object Constraint Language [56] es un lenguaje para la descripción formal de expresiones en los modelos UML. Sus expresiones pueden representar invariantes, pre-condiciones, post-condiciones, inicializaciones, guardas, reglas de derivación, así como consultas a objetos para determinar su estado.

Su papel principal es el de completar los diferentes artefactos de la notación UML con requerimientos formalmente expresados; la versión OCL2.0 (Object Constraint Language 2.0) fue adoptado en octubre de 2003 por el grupo OMG como parte de UML 2.0.

ODM

ODM (Ontology Definition Metamodel[4]) es un metamodelo de definición de ontologías que pertenece a la familia de metamodelos de MOF[5]. Permite tanto la transformación entre dichos metamodelos como desde ellos a UML y viceversa. Los metamodelos que incluye el ODM reflejan una sintaxis abstracta para representar el conocimiento de forma estándar y lenguajes de modelado conceptual

OWL

Ontology Web Language [68] es un lenguaje de marcado construido sobre RDF y codificado en XML para publicar y compartir datos usando ontologías [157] en la web.

OWL tiene tres variantes:

- *OWL Lite*: soporta la clasificación de jerarquías y definición de restricciones simples.

[3] `http://www.omg.org/spec/OCL/2.0`
[4] `http://www.omg.org/spec/ODM/1.0/PDF`
[5] `http://www.omg.org/mof/`

- *OWL DL*: permite una mayor expresividad reteniendo completitud computacional en donde se asegura que todas las conclusiones son computables y decidibles en tiempo finito.
- *OWL Full*: Máximo nivel de expresividad sin garantía de computabilidad.

Estas variantes incorporan diferentes funcionalidades, y en general, *OWL Lite* es más sencillo que *OWL DL* y *OWL DL* es más sencillo que *OWL Full*.

OWL Lite está construido de tal forma que toda sentencia pueda ser resuelta en tiempo finito. La versión más completa de *OWL DL* puede contener 'bucles' infinitos.

OWL DL se basa en la lógica de descripción **{SHOIN} (D)**. El subconjunto *OWL Lite* se basa en la lógica menos expresiva **{SHIF} (D)**.

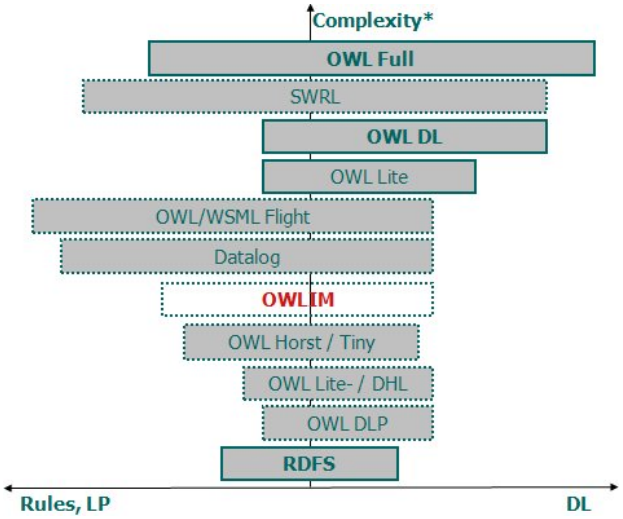


Figura C.3.: OWL en el contexto de DL

A su vez, *OWL* forma parte del siguiente stack de tecnologías relacionadas con lo que es la Web Semántica, detallado según Figura C.4.

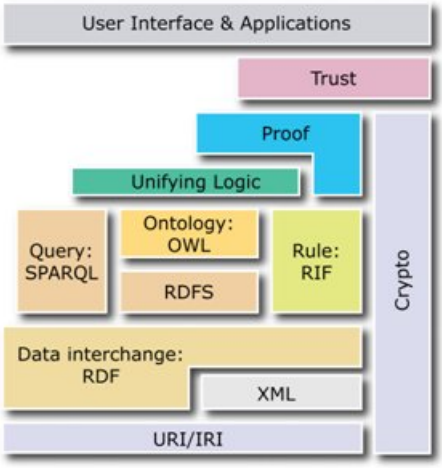


Figura C.4.: Stack de tecnologías en la Web Semántica

OWL 2

OWL 2 refina y extiende OWL; la especificación alcanzó el estatus de candidato a recomendación en junio de 2009.

OWL 2 corrige varios problemas de la especificación inicial y agrega expresividad al lenguaje acorde a las últimas investigaciones relacionadas con *Description Logics*. Además, agrega la posibilidad de utilizar XML y otros formatos compactos de representación para el intercambio de ontologías. Finalmente, OWL 2 ofrece 3 subconjuntos del lenguaje denominados perfiles:

- *OWL 2 EL* es particularmente útil para ontologías con gran cantidad de ítems y propiedades; el razonamiento es computable en tiempo polinómico.
- *OWL 2 QL* es de interés para ontologías con una importante cantidad de datos en donde el mecanismo de consulta puede referenciar bases de datos relacionales. Se basa en *DL-Lite*.
- *OWL 2 RL* es un fragmento de OWL 2 que permite el uso de tecnologías basadas en reglas.

C.2. Propietarios

A nivel comercial, cada motor de evaluación de reglas implementa sus reglas de negocio de la manera que más le convenga para soportar sus funcionalidades. Existen diferentes formas de especificar las reglas de negocio con sus diferentes lenguajes, como por ejemplo:

- Orientados a objetos como es el caso de *Drools*, *JRules*, *Blaze Advisor*, *Oracle Business Rules*.
- Web semántica como es el caso de *JenaRules*.
- Inteligencia Artificial utilizando frames como es el caso de *Jess* y *Clips*.
- Internal DSL como es el caso de *RuleBy*.

Se realiza un resumen de los principales lenguajes de definición de reglas de negocio propietarios.

RL

Rule Language, lenguaje propietario de Oracle para definir reglas de negocio específico para trabajar desde el ambiente Java.

Las reglas siguen una estructura *If-Then* en donde se definen las condiciones y acciones respectivamente.

Ejemplo:

```
rule driverAge
{
  if (fact Driver d1 && d1.age < 16)
  {
  println("Invalid Driver");
  }
}
```

A partir de la versión 11g del producto Middleware Oracle Fusion, se agregan interfaces de alto nivel para poder definir las reglas.

Por ejemplo definiendo una regla se detalla la interfaz de la Figura C.5.

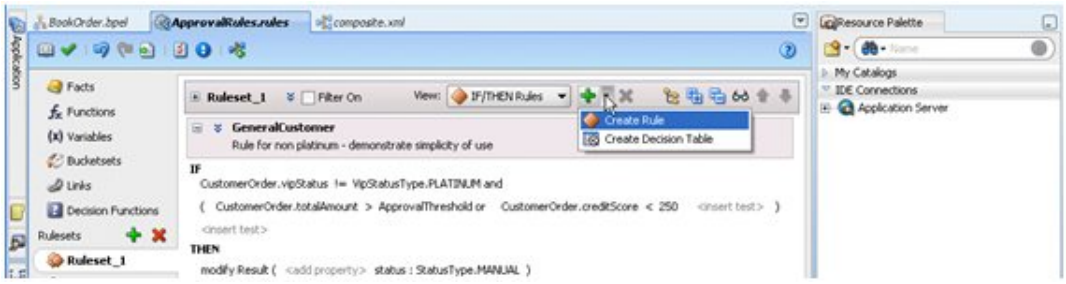


Figura C.5.: Oracle Front-end

SRL

Structured Rules Language, es un lenguaje propietario basado en XML desarrollado por la empresa FICO utilizado en su producto Blaze Advisor.

Es un lenguaje orientado a objetos diseñado con el objetivo de lograr escribir y leer reglas de negocio de forma "similar" al inglés. Una regla tipo se detalla a continuación:

```
If
  the applicant for the loan is a current bank customer
  and has a checking account with direct paycheck deposit
  and has a balance greater than $10,000
then
  lower the interest rate by 1%.
```

DRL

Drools Rule Language, lenguaje de definición de reglas utilizado por la herramienta Drools. Las reglas siguen una estructura *When-Then* en donde se definen las condiciones y acciones respectivamente.

Ejemplo:

```
rule "Check Patient for Heart Attack"
when
  $patient : Patient(
    breathing == false ,
    pulse == false)
    (face == blue or face == white)
  )
then
    $hospital.callForAssistance();
    $patient.startCPR();
end
```

La Figura C.6 detalla un ejemplo definiendo una regla desde el Front-end Web.

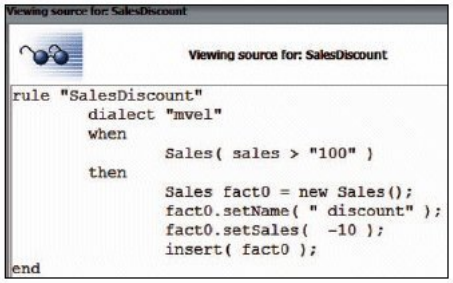


Figura C.6.: Drools Front-end

JessML

Es el lenguaje propietario de definición de reglas utilizado por Jess. Primero se definen los objetos y propiedades sobre los cuales se desea trabajar y posteriormente se definen las reglas en base a dichos objetos.

Se utiliza la cláusula *deftemplate* para definir los slots (propiedades) de los hechos sobre los cuales se quiere trabajar. El siguiente ejemplo define un hecho con 3 propiedades (slots):

```
deftemplate person "People in acturial database"
  (slot name (default OCCUPANT))
    (slot age)
    (slot gender)
    (multislot hobbies))
Jess>(assert
  (person (age 22)
    (name "Jane Doe")
    (gender Female)
    (hobbies snowboarding "restoring antiques")
  ))
```

Jess ofrece 3 tipos diferentes de hechos, cada uno con su estructura, índice y uso propio.

- *unordered facts*: es el equivalente a un registro en una tabla relacional, con campos individuales que se corresponden con las columnas de una tabla.
- *ordered facts*: carece de la estructura con campos nombrados como el caso anterior, es solo una lista plana y se utiliza cuando no se necesita una estructura.
- *shadow facts*: son hechos desordenados que están relacionados a objetos Java de una aplicación; permite razonar sobre eventos que ocurren fuera del ambiente *Jess*.

Ejemplo de una regla:

```
Jess>( defrule cambiar-bebe-si-esta-mojado
  ?mojado <- (bebe-mojado)
  =>
  (cambiar-bebe)
  (retract ?mojado)
)
TRUE
```

BAL

Business Action Language, lenguaje de reglas de negocio de alto nivel de abstracción utilizado por la herramienta JRules. Mapea un conjunto de verbos y sustantivos usado por los analistas de negocio en reglas que utilizan las clases, métodos y campos del modelo de objetos del negocio.

Ejemplo:

```
If
    the shopping cart value is greater than $ 100
    and the customer category is gold
then
    Apply a 15% discount
```

IRL

Ilog Rule Language representa el formato ejecutable del lenguaje de reglas utilizado por ILOG.

Ejemplo:

```
rule latestBankruptcy1 {
  property task = "computation";
  when {
    borrower: Borrower(hasLatestBankrupcy()) from borrower;
    loan: Loan() from loan;
    evaluate (age: DateUtil.getAge(borrower.getLatestBankruptcyDate(), loan.getStartDate(
    evaluate (age <=1);
  }
  then {
    report.addCorporateScore(-200);
  }
}
```

Un ejemplo editando una regla desde el front-end se detalla en la Figura C.7.

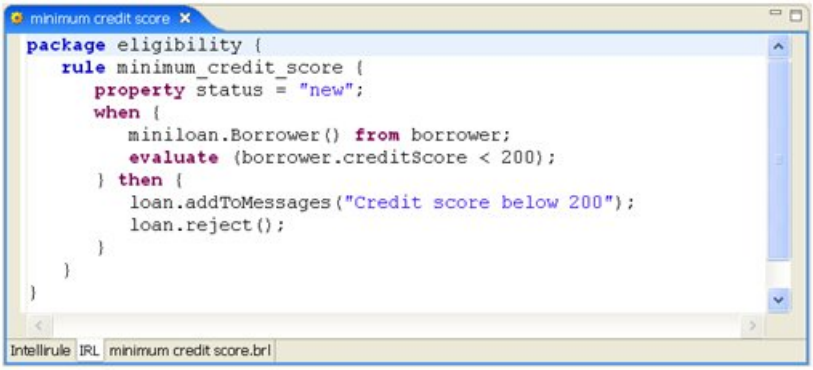


Figura C.7.: JRules Front-end

BRL

Business **R**ule **L**anguage, representa el formato de definición de reglas utilizado por ILOG específico para el experto del negocio.

Ejemplo:

```
If
  the credit score of 'the borrower' is less than 200
then
  add "Credit score below 200 " to the messages of 'the loan';
  reject 'the loan';
```

Un ejemplo editando una regla desde el administrador de reglas se detalla en la Figura C.8.

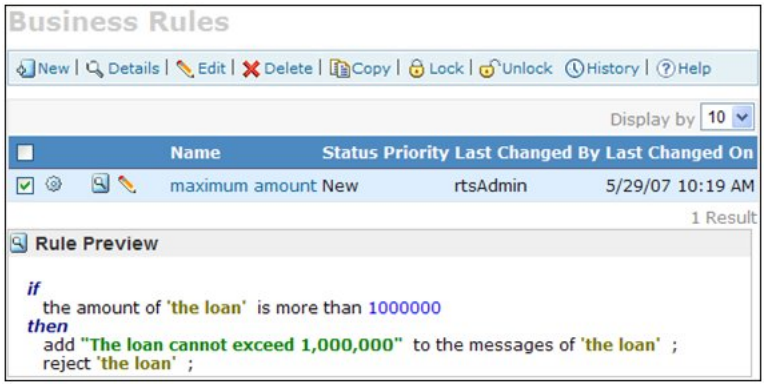


Figura C.8.: JRules Front-end utilizando interfaz Web

Jena Rules

JenaRules se basa en RDF(S) y utiliza la representación en tripletas definido por RDF (también permite especificar las reglas en notación N3 y Turtle), forma parte de la herramienta Jena [40]. Para poder trabajar con JenaRules es necesario transformar los objetos del negocio a representación RDF para poder utilizar los motores de inferencia provistos.

Una clase Java mapea a la definición *rdfs:Class*; mientras que las propiedades de las clases mapean a la propiedad *rdf:Property*. Las propiedades RDFS *rdfs:domain* y *rdfs:range* especifican la *clase* y *tipo* de propiedad.

Supongamos se tienen las clases *Person* y *Driver* con la siguiente especificación:

```
public class Person {
  private String name;
  private int age;
}
```

```
public class Driver extends Person {
  private int accidentsNumber;
}
```

La representación RDF es la siguiente:

```
<rdfs:Class rdf:about="Person" />
<rdfs:Class rdf:about="Driver">
  <rdfs:subClassOf rdf:resource="#Person"/>
</rdfs:Class>
<rdf:Property rdf:about="Person.name">
  <rdfs:range rdf:resource="xs:string"/>
  <rdfs:domain rdf:resource="#Person"/>
</rdf:Property>
<rdf:Property rdf:about="Person.age">
  <rdfs:range rdf:resource="xs:int"/>
  <rdfs:domain rdf:resource="#Person"/>
</rdf:Property>
<rdf:Property rdf:about="Driver.accidentsNumber">
  <rdfs:range rdf:resource="xs:int"/>
  <rdfs:domain rdf:resource="#Driver"/>
</rdf:Property>
```

Un ejemplo de una regla que utiliza las clases detalladas anteriormente es la siguiente:

```
@prefix eg: http://com.sample/
# Hello World
[HelloWorld:(?m rdf:type eg:Person)
            (?m eg:name 'Miguel'^^xsd:string)
            (?m eg:age ?s)
            →
            (?m eg:name 'Michael^^xsd:string)
            log(?m, 'matched')
```

que define que cuando exista una persona de nombre “Miguel”, se registra el caso a través de un mensaje de log.

RuleBy

Ruleby utiliza un DSL interno (Internal DSL) para poder especificar las reglas de negocio. Esto quiere decir que la manipulación de los objetos de negocio es transparente tratándose de los propios objetos Ruby.

Un ejemplo de una regla:

```
class HelloWorldRulebook < Rulebook
  def rules
    rule [Message, :m, m.status == :HELLO] do |v|
      puts v[:m].message
```

```
    v[:m].message = "Goodbye world"
    v[:m].status = :GOODBYE
    modify v[:m]
  end
end
```

D. Algoritmos más importantes

En este apéndice se explican los algoritmos más importantes que utilizan los motores de evaluación de reglas. Se hace especial énfasis al algoritmo *Rete* dado que es un algoritmo de dominio público muy utilizado tanto en productos comerciales como open-source.

D.1. Algoritmos de Inferencia

Las reglas procesadas por un algoritmo de inferencia son definidas en forma independiente entre sí.

Al contrario que los algoritmos secuenciales, en donde las relaciones entre las reglas se definen o calculan en forma explícita, cuando se utiliza la inferencia se determinan las dependencias entre reglas utilizando datos.

Los datos gobiernan el camino de reglas a ejecutar; se puede ejecutar más de una vez una misma regla en caso que alguna regla cambie datos que afecten las condiciones de otras.

Los desarrolladores tienen la ventaja de diseñar reglas en forma totalmente independiente, mientras que la interacción entre las mismas se encuentra administrada por los datos de sesión durante la ejecución.

Como desventaja, en caso de grandes conjuntos de reglas, la performance puede ser baja y las reglas pueden interactuar de una manera inesperada.

Algunos de los algoritmos más utilizados son:

- *Rete*.
- *ReteII*.
- *ReteIII*.
- *ReteOO*: Rete Object-Oriented.
- *Rete Plus*.
- *DRETE*: Distributed Rete.
- *TREAT*: Tree Associative Temporal redundancy.

- *LEAPS*: Lazy Evaluation Algorithm for Production Systems.
- *DETI*: Design-Time Inferencing.
- *MatchBox*.
- *HAL*: Heuristically-Annotated-Linkage match algorithm.

Rete

Rete, es un algoritmo diseñado por Charles L. Forgy [150] en 1972 para eliminar las ineficiencias del "matcheo" de patrones (pattern matching). Es decir, el pattern-matching es costoso computacionalmente ya que se debe decidir si una lista arbitraria coincide con otra y depende del tamaño y número de variables en la lista.

El algoritmo *Rete*; elimina dos fuentes principales de ineficiencia:

- las reglas generalmente comparten condiciones y es ineficiente tratar de "matchear" todas las condiciones con los elementos.
- la memoria de trabajo cambia poco en cada ciclo y es ineficiente revisar nuevamente todos los elementos ante cualquier actualización.

Para que las operaciones sean más eficientes, el algoritmo construye una red ordenada en forma de árbol que contiene todos los patrones en las condiciones (sólo una vez) y se utiliza para revisar contra la memoria de trabajo evitando ciclos internos guardando los resultados anteriores de las evaluaciones de los hechos en cada iteración. Solo se testean elementos nuevos o modificados en la memoria de trabajo.

Rete deriva del latín red, y es implementado construyendo una red de nodos interconectados; cada nodo representa uno o más tests a realizar en la parte derecha (RHS – Right Hand Side) de una regla.

Para procesar hechos y reglas, el algoritmo *Rete* crea y utiliza un nodo de entrada para cada definición de un hecho y un nodo de salida por cada regla. Los hechos referencian el flujo de los nodos de entrada a los de salida. Entre los nodos de entrada y salida están los nodos de tipo *test* y *join*. Un test ocurre cuando una condición de una regla tiene una expresión booleana. Un *join* ocurre cuando una condición de una regla utiliza el operador *AND* con dos hechos. Una regla se activa cuando su nodo de salida referencia hechos; estas referencias son cacheadas a lo largo de la red para aumentar la velocidad de reproceso de las reglas activadas. Cuando un hecho es agregado, modificado o eliminado, la red *Rete* actualiza el cache y las activaciones de reglas; sólo requiere un trabajo incremental.

Cada nodo tiene una o dos entradas y un número ilimitado de salidas. Al agregar\quitar de la memoria de trabajo diferentes hechos, los mismos son "procesados" por esta red de nodos. Los nodos de entrada son los principales de la

red (top) mientras que los de salida están al final (bottom). Al inicio de la red, los nodos de entrada separan los hechos en categorías de acuerdo a su cabeza (head). A medida que se desciende de nivel en el árbol se encuentran discriminaciones y asociaciones entre hechos más finas hasta llegar al final de la red donde los nodos representan reglas individuales.

Cuando un conjunto de hechos son filtrados y se llega al final de la red quiere decir que se han pasado todos los test sobre el lado izquierdo de las reglas (LHS – Left Hand Side), por lo cual se convierte en un nuevo registro de activación o un comando para cancelar una regla de activación anterior, es decir se encontraron un conjunto de reglas que "matchean" con el patrón.

Este algoritmo provee los siguientes beneficios:

- *Independencia del orden de reglas*: las reglas se pueden agregar y eliminar sin afectar otras reglas.
- *Optimización entre múltiples reglas*: reglas con condiciones comunes comparten nodos en la red.
- *Alta performance en los ciclos de inferencia*: cada gatillado de cada regla cambia típicamente unos pocos hechos y el costo de actualizar la red es proporcional al número de hechos que cambiaron y no al número total de hechos y reglas.

ReteII

Algoritmo propietario[1] sucesor de *Rete* inventado por Dr. Charles Forgy que ejecuta entre 100 y 1000 veces más rápido que el original dependiendo de la complejidad de las reglas y objetos que se ingresan a la memoria de trabajo.

Rete ejecuta en forma eficiente cuando se trata de un número importante de reglas. Sin embargo, cuando la cantidad de objetos en memoria aumenta, disminuye la performance. *ReteII* realiza optimizaciones en este sentido.

ReteIII

Rete III[2] es un algoritmo propietario desarrollado por Charles Forgy que parte del algoritmo *ReteII* agregando RLT (Relational Logic Technology[3]). Fue desarrollado especialmente para hacer posible que los motores de reglas puedan manejar efectivamente un conjunto importante de datos así como también grandes cantidades de reglas.

El algoritmo *Rete III* forma parte del motor de evaluación de reglas *Blaze Advisor*.

[1] `http://www.pst.com/rete2.htm`
[2] `http://www.edmblog.com/weblog/2005/09/what_is_rete_ii.html`
[3] `http://www.faqs.org/patents/app/20090106063`

ReteOO

ReteOO (Object Oriented Rete) es una implementación orientada a objetos de *Rete*[4]. Fue adaptado por Bob McWhirter y es utilizado en *Drools*.

Rete Plus

Rete Plus es una implementación del algoritmo *Rete* por la empresa *IBM ILOG* agregando algunas extensiones.

DRETE

DRete (Distributed *Rete*) es un algoritmo desarrollado por Michael A. Kelly y Rudolph E. Seviora [168] de "matcheo" de patrones.

En *DRete* la fase de "matcheo" de patrones se particiona y paraleliza al nivel de comparación de tokens.

TREAT

Daniel P. Miranker en 1989 modifica el algoritmo *Rete* al cual le denomina *TREAT* (TREe Associative Temporal redundancy), combinándolo con técnicas de optimización derivadas de los motores de bases de datos relacionales.

TREAT [181] no utiliza una red discriminada de reglas y no cachea los resultados intermedios en las intersecciones de las redes tal como lo hace *Rete*; el matching de patrones se recalcula cada vez que se necesita. Esto quiere decir que este algoritmo es una alternativa a *Rete* cuando el consumo de memoria es un requerimiento a tener en cuenta.

LEAPS

LEAPS (Lazy Evaluation Algorithm for Production Systems [124]) es una extensión del algoritmo *TREAT*.

El aporte fundamental de este algoritmo es que las tuplas se evalúan solo cuando se necesitan (lazy evaluation).

MatchBox

Match Box [202] es un algoritmo desarrollado por Mark Perlin de "matcheo" de patrones incremental para determinar la instanciación de tuplas en sistemas basados en reglas de inferencia hacia adelante.

[4] `http://legacy.drools.codehaus.org/ReteOO`

DETI

DETI (Design-Time Inferencing) es un algoritmo de "matcheo" de patrones propietario desarrollado por la empresa Corticon.

La mayor parte del chequeo de reglas se realiza en tiempo de diseño y no en ejecución. Esto elimina el problema que puede ocurrir en ejecución cuando existe razonamiento circular, reglas duplicadas o inconsistentes e incompletitud en el juego de reglas definido.

En comparación con *Rete*, tiene la contraparte que pierde la facilidad de eliminar o modificar una regla "on-the-fly".

HAL

HAL (Heuristically-Annotated-Linkage) [174] es un algoritmo que presenta una solución diferente para el "matcheo" de patrones ya que utiliza el conocimiento heurístico embebido en las reglas y los elementos que conforman la base de hechos y los relaciona.

D.2. Algoritmos Secuenciales

Cuando se utilizan los algoritmos secuenciales las reglas se conectan explícitamente unas a otras dentro de su propia definición. De esta manera los desarrolladores pueden llevar la traza de ejecución de un grupo de reglas.

La ventaja principal de este tipo de algoritmos es su velocidad, dado que los desarrolladores explícitamente determinan la secuencia de reglas dejando un diseño totalmente transparente. Por otro lado, no se provee la flexibilidad del diseño de reglas en forma independiente, como es el caso cuando se utilizan algoritmos de inferencia.

Algunos ejemplos de herramientas que disponen este modo de ejecución son: Blaze Advisor [21], JRules [107] y Windows Workflow Foundation [109].

D.3. Benchmarks

Existen benchmarks[5] específicos para analizar y comparar la performance de los motores de evaluación de reglas.

Algunos ejemplos son:

- Miss Manners.
- Waltz.

[5] `http://illation.com.au/benchmarks/`

- WaltzDB.
- ARP.
- Weaver.

Miss Manners

El benchmark *Miss Manners* forma parte de un conjunto de benchmarks recolectados por el grupo del Dr. Daniel Miranker[6] en la Universidad de Texas [130].

El benchmark *Manners* trata sobre la asignación de asientos a los invitados en una cena. El criterio de asignación de asientos tiene en cuenta el interés de las diferentes personas y su sexo. Se trata de ubicar a las personas de diferentes sexos y con aficiones comunes juntas. La carga de trabajo del programa que resuelve la asignación de asientos depende de la cantidad de invitados a la fiesta; generalmente se ejecuta con valores que van desde los 32 a los 96 huéspedes.

Manners va aumentando el número de elementos en la memoria de trabajo (hechos) sobre las cuales se aplican las reglas. En el caso mas simple, la memoria de trabajo tiene un promedio de 400 hechos hasta llegar a un problema más grande en donde se trabaja con más de 2700 hechos. Al aumentar la cantidad de hechos aumenta en consecuencia la cantidad de activaciones de reglas por cada iteración del motor de evaluación de reglas.

Manners realiza una búsqueda en profundidad para resolver la asignación de asientos a los diferentes invitados según las restricciones detalladas anteriormente. Ha sido diseñado para ejercitar *joins* entre diferentes objetos (cross product joins) y actividades de la *Agenda*.

Waltz y WaltzDB

El benchmark *Waltz* fue desarrollado en la Universidad de Columbia. Se trata de un sistema experto diseñado para ayudar en la interpretación en tres dimensiones de un dibujo de dos dimensiones. A partir de una colección de líneas que pueden encontrarse en un detector de ejes para el reconocimiento visual; se trata de etiquetar todas las líneas de la imagen basándose en la propagación de restricciones. Sólo imágenes que contengan uniones de dos y tres líneas están permitidas.

El conocimiento que utiliza *Waltz* está embebido en las reglas. La propagación de restricciones consta de 17 reglas que asignan etiquetas a las líneas basándose en etiquetas ya existentes. Además, existen 4 reglas que establecen las etiquetas iniciales.

[6] `http://www.cs.utexas.edu/ftp/pub/ops5-benchmark-suite/`

El benchmark *Waltzdb* desarrollado en la Universidad de Texas extiende a *Waltz* ya que amplía el programa para manejar casos mas generales en donde se trabaja con dibujos que involucran la composición de 4 a 6 líneas.

El método utilizado para resolver el problema de etiquetado es una versión del algoritmo descrito en [226]. La diferencia clave entre ambos benchmarks es que en *Waltzdb* se utiliza una base de datos de las etiquetas de línea legales que se aplican a las uniones de una manera limitada mientras que *Waltz* utiliza un conjunto fijo de restricciones según detallan las 21 reglas de restricción.

ARP

El benchmark *ARP* (Aeronautical Route Planner) es un planificador de ruta para un robot aéreo que utiliza el algoritmo de búsqueda *A** [140] para lograr un costo mínimo.

ARP calcula el costo más bajo teniendo en cuenta el terreno, diferentes amenazas y datos de costos. La ruta intentará evitar distintos tipos de amenazas como por ejemplo misiles tierra-aire.

ARP minimiza tres costos para una ruta: el costo que lleva viajar la distancia planteada, el costo de mantenerse a una altura especifica y el costo de estar sobre la tierra.

El algoritmo de búsqueda *A** es utilizado para restringir la porción del espacio de búsqueda que se debe examinar para calcular la ruta. Los datos de entrada es una base de datos de información sobre el terreno.

Weaver

Weaver es un sistema experto diseñado para realizar ruteo de canales en integración de sistemas muy grandes (VLSI – Very Large Scale Integration Systems[7]) y box routing [165]. Es un programa grande y complejo que se compone de varios sistemas expertos independientes que se comunican a través de un pizarrón (blackboard) [172].

[7] `http://www.cs.cmu.edu/~jab/pubs/propo/node19.html`

E. Evaluación de Herramientas

Tomando como referencia el análisis de herramientas existentes en el mercado detallado en el Anexo B se analiza al menos una herramienta para las principales plataformas de ejecución GeneXus: *Java*, *.Net* y *Ruby*; investigando posibles mecanismos de integración con GeneXus.

La evaluación y posterior selección de herramientas consistió de las siguientes etapas:

- *Pre-selección*: Seleccionar una lista de herramientas candidatas a partir del estudio de herramientas detallado en el Anexo B priorizando aquellas herramientas de código abierto.
- *Evaluación*: Desarrollar los casos de uso que detalla el Anexo G utilizando código generado GeneXus.
- *Selección*: Seleccionar aquellas herramientas que logren resolver los casos de uso planteados.

E.1. Pre-selección

La etapa de pre-selección implica obtener al menos una herramienta candidata por plataforma de ejecución GeneXus para poder pasar a la siguiente fase que se corresponde con la evaluación de la herramienta en sí según los escenarios planteados.

A partir del universo de herramientas detallado, se ordenan por plataforma y tipo (open-source y comercial) y analizan teniendo en cuenta:

- Organización o Empresa que respalda la herramienta.
- Documentación y ejemplos existentes.
- Características principales de la herramienta.
- Movimiento en foros de consulta.
- Versiones liberadas.

Java

En la plataforma *Java* es donde se registra la mayor cantidad de herramientas open source. Para la pre-selección se analizan las herramientas *JEOPS*, *JLisa*, *Jena Rules*, *Hammurapi Rules*, *JRule Engine*, *CommonRules*, *JxBRE*, *Mandarax*, *Prova*, *Drools*, *Open Rules*, *SweetRules* y *Jess*.

Las herramientas *JEOPS* [43] y *JLisa* [45] no registran actualizaciones de sus productos desde el 2003.

Jena Rules [41] forma parte de la herramienta *Jena* con foco específico en la Web Semántica originario de HP Labs. Incluye implementaciones de algoritmos de tipo forward y backward chaining; es un proyecto que se encuentra activo y existe una comunidad importante utilizándola.

Las herramientas *Hammurapi Rules* [29] y *JRule Engine* [46] sólo proveen el estándar *JSR-94* [36] como mecanismo de comunicación con el motor de reglas. Según el análisis de estándares realizado en el capítulo 2, el uso de dicho estándar no prosperó.

Las herramientas *CommonRules* [7], *JxBRE* [48] y *Mandarax* [51] fueron descontinuados. El proyecto *CommonRules* fue cancelado por IBM en el 2003, *JxBRE* descontinuado en el 2004 y *Mandarax* no registra actividad ni actualizaciones desde agosto de 2006.

Prova [70] se desarrolla a partir de *Mandarax* y combina los beneficios de la programación declarativa con la orientación a objetos. No se dispone de buena información sobre el funcionamiento del motor de evaluación de reglas.

Drools [13] es uno de los motor de reglas más utilizados contando con el respaldo de la empresa *RedHat*. Dispone una amplia documentación y un foro muy activo. Registra actividades tanto en la industria como en la academia.

Open Rules [62] no tiene capacidad de inferencia, simplemente ejecuta un grupo de reglas definidos en planillas de cálculo o tablas de decisión.

SweetRules [97] tiene foco en la Web Semántica y no registra nuevas versiones desde abril de 2005.

Jess [44] es un motor de reglas que hace tiempo existe en el mercado desarrollado en los laboratorios Sandia National Laboratories. Inicialmente derivó de Clips [11] y se reescribe en Java. Es muy utilizado en el ambiente académico registrándose muy buena documentación en línea además de una versión de evaluación.

Se pre-seleccionan las herramientas *Drools*, *Jena Rules* y *Jess*.

.Net

Para la plataforma *.Net* se analizan las herramientas *NxBRE*, *Drools.NET*, *Expergent* y *Simple Rule Engine*.

Simple Rule Engine [95] no registra movimientos desde junio de 2006 con muy poco movimiento en los foros existentes.

Expergent [16] tampoco registra movimientos a nivel de código del proyecto, el último commit realizado al repositorio del proyecto se registra en julio de 2008[1].

La herramienta *NxBRE* [55] registra más movimiento que los casos anteriores con documentación relacionada de cómo utilizar el motor según los diferentes modos de ejecución. El proyecto fue actualizado en enero de 2009.

Drools.NET [14] es un port a la plataforma .Net de la herramienta *Drools 3.0* específica para Java. Al ser un port de la plataforma se puede reutilizar la extensa documentación existente desde la versión Java.

Se pre-seleccionan las herramientas *NxBRE* y *Drools.NET*.

Ruby

Para *Ruby* se encontraron las herramientas open source *Rools*, *Ruleby*, *SIE* y *Khammurabi*.

SIE [94] es un utilitario desarrollado por Peter Hickman que ofrece capacidades de inferencia básicas no existiendo prácticamente documentación de cómo utilizarla.

La gema *Khammurabi* [49] prácticamente no registra movimiento desde agosto de 2005.

Ruleby [85] provee un DSL [169] para definir reglas de negocio. El motor de inferencia utiliza el algoritmo Rete totalmente implementado en Ruby. En el 2008 en la conferencia *Ruby Hoedown*[2] se realiza una presentación de la gema. A partir de dicha presentación, se registra un aumento en el interés sobre la herramienta a través de los foros de consulta asociados.

Accediendo al sitio del proyecto *Rools* [84] se verifica que la gema prácticamente no se ha utilizado desde julio de 2007 cuando se publica la versión 0.3. El foro público consta de 4 mensajes[3] intercambiados a la fecha.

En forma automática, para *Ruby* se pre-selecciona la herramienta *RuleBy*.

E.2. Evaluación

La estrategia de evaluación consistió de los siguientes pasos:

[1] `http://code.google.com/p/expergent/source/list`
[2] `http://www.bestechvideos.com/2008/08/21/ruby-hoedown-2008-ruleby-the-rule-engine-for-ruby`
[3] `http://rubyforge.org/forum/?group_id=1166`

- Se instala el motor de reglas y estudia el juego de ejemplos incluidos en caso que exista. Esto incluye la especificación del modelo de objetos, reglas, hechos y la interacción desde el lenguaje procedural con el motor correspondiente.
- Se realiza la primera prueba de concepto, se implementa el ejemplo "*Hola Mundo*" (detallado en la sección G.2 del Anexo G) para asegurarse que las funcionalidades básicas son soportadas por el motor. En caso que la evaluación sea satisfactoria se continúa con los casos de uso relacionados al ejemplo GeneXus, de lo contrario se selecciona el siguiente motor a evaluar.
- Se genera el escenario de prueba GeneXus en los diferentes lenguajes: *Java*, *C#* y *Ruby*, según corresponda.
- Se realizan pruebas de integración con el código generado GeneXus para los escenarios *Cliente*, *Producto* y *Factura*; sección G.2 del Anexo G. En este caso implica:
  - Instalar un ambiente de prueba para cada generador Web GeneXus.
  - Generar procedimientos de carga de prueba con datos genéricos.
  - Integrar código GeneXus para que se dialogue con el motor de reglas correspondiente.
  - Definir los artefactos necesarios para especificar las reglas definidas en el escenario.
  - Programar la interacción entre el código procedural y el motor de reglas.
  - Verificar que los objetos GeneXus resultan afectados a partir de la evaluación de reglas ejecutada.
- Se resume el resultado de la evaluación.

Drools

Se realizan las pruebas correspondientes para todos los casos de uso usando *Drools 5.0* [13].

Escenario 0 - Hello World

Se define el siguiente juego de reglas en el lenguaje del motor de reglas *Drools Rule Language* (DRL).

```
package com.sample
import com.sample.DroolsTest.Message;

rule "Hello World"
  when
```

```
    m:Message( status == Message.HELLO, message:message )
  then
    System.out.println( message );
    m.setMessage("Goodbye cruel world");
    m.setStatus( Message.GOODBYE );
    update( m );
end

rule "GoodBye"
  when
    Message( status == Message.GOODBYE, message:message )
  then
    System.out.println( message );
end
```

Se evalúa con el siguiente código procedural:

```
Message message = new Message();
message.setMessage("Hello World");
message.setStatus(Message.HELLO);
workingMemory.insert(message);
workingMemory.fireAllRules();
```

El funcionamiento es correcto según lo especificado ya que al ejecutarse el ejemplo desde la consola se detallan los siguientes mensajes:

```
Hello World
Goodbye cruel world
```

Notar que la sintaxis para definir reglas en este caso implica utilizar el lenguaje de expresiones MVEL [54] en la sección de condiciones y código Java en la sección de consecuencias.

Integración GeneXus Java - Drools

En *Drools* la base de hechos (fact model) sobre la cual trabaja el motor son las propias clases Java preferentemente POJOs que implementen la interfaz JavaBeans [38]. En este caso para satisfacer dicho requerimiento se utiliza un *Business Component* GeneXus.

A partir de los fuentes GeneXus generados para el caso de uso, se agrega un método que llama a una clase externa que realiza la llamada e interacción con el motor de reglas pasando como parámetro el correspondiente *Business Component*. El código extra agregado es el siguiente:

```
mybr01.setCustomerBr01(customerbrinteraction01.this.AV8myCusto);
```

y el código de ejemplo que interactúa con el motor es:

```
//load up the rulebaseKnowledgeBase
ruleBase = readRule("/Customer01.drl");
StatefulKnowledgeSession workingMemory = ruleBase.newStatefulKnowledgeSession();
insertInWM(myCustomer01, workingMemory);
fireRules(workingMemory);
```

Escenario 1 - Cliente

Se implementa un juego de reglas básico para asegurarse que se detectan objetos de tipo `Customer`.

```
package com.sample
dialect "mvel"

import com.sample.SdtCustomer;
rule "Male Customer"
  when
    c : SdtCustomer( gxTv_SdtCustomer_Customergender == "M" )
  then
    System.out.println("Customer is male");
end

rule "Customer"
  salience 10
  when
    m : SdtCustomer( )
  then
    System.out.println("Customer");
end
```

Resultados obtenidos al ejecutarse con un cliente masculino:

```
Customer is male
Customer
```

Se implementan los juegos de reglas especificados en el escenario.

```
package com.sample
dialect "mvel"
import com.sample.SdtCustomer;

rule "Todos los clientes son B"
  no-loop true
  when
    c : SdtCustomer( )
  then
    System.out.println("Todos los clientes son B");
    c.setgxTv_SdtCustomer_Customertypeid( 2 ); // B
    update( c );
end
```

```
rule "Clientes masculinos de hasta 25 años son de tipo A."
  no-loop true
  when
    c : SdtCustomer(
        gxTv_SdtCustomer_Customergender == "M",
        gxTv_SdtCustomer_Customerage <= 25 )
  then
    System.out.println("Clientes masculinos de hasta 25 años son de tipo
A");
    c.setgxTv_SdtCustomer_Customertypeid( 1 ); // A
end

rule "Clientes femeninos de hasta 30 años son de tipo A."
  no-loop true
  when
    c : SdtCustomer(
        gxTv_SdtCustomer_Customergender == "F",
        gxTv_SdtCustomer_Customerage <= 30 )
  then
    System.out.println("Clientes femeninos de hasta 30 años son de tipo
A");
    c.setgxTv_SdtCustomer_Customertypeid( 1 ); // A
end

rule "Clientes de mas de 70 años son de tipo C."
  no-loop true
  when
    c : SdtCustomer( gxTv_SdtCustomer_Customerage > 70 )
  then
    System.out.println("Clientes de mas de 70 años son de tipo C");
    c.setgxTv_SdtCustomer_Customertypeid( 3 ); // C
end
```

y el segundo juego de reglas:

```
package com.sample
dialect "mvel"
import com.sample.SdtCustomer;

rule "Todos los clientes son B"
  no-loop true
  salience 100 // aumento prioridad
  when
    c : SdtCustomer( )
  then
    System.out.println("Todos los clientes son B");
    c.setgxTv_SdtCustomer_Customertypeid( 2 ); // B
end

rule "Clientes masculinos de hasta 20 años son de tipo A."
```

```
  no-loop true
  when
    c : SdtCustomer(
        gxTv_SdtCustomer_Customergender == "M",
        gxTv_SdtCustomer_Customerage <= 20 )
  then
    System.out.println("Clientes masculinos de hasta 20 años son de tipo
A");
    c.setgxTv_SdtCustomer_Customertypeid( 1 ); // A
end

rule "Clientes femeninos de hasta 30 años son de tipo A."
  no-loop true
  when
    c : SdtCustomer(
        gxTv_SdtCustomer_Customergender == "F",
        gxTv_SdtCustomer_Customerage <= 30 )
  then
    System.out.println("Clientes femeninos de hasta 30 años son de tipo
A");
    c.setgxTv_SdtCustomer_Customertypeid( 1 ); // A
end

rule "Clientes de mas de 70 años son de tipo C."
  no-loop true
  when
    c : SdtCustomer( gxTv_SdtCustomer_Customerage > 70 )
  then
    System.out.println("Clientes de mas de 70 años son de tipo C");
    c.setgxTv_SdtCustomer_Customertypeid( 3 ); // C
end

rule "Clientes entre 50 y 60 años asignar el tipo C."
  no-loop true
  when
    c : SdtCustomer(
        gxTv_SdtCustomer_Customerage >= 50,
        gxTv_SdtCustomer_Customerage <= 60 )
  then
    System.out.println("Clientes entre 50 y 60 años asignar el tipo C");
    c.setgxTv_SdtCustomer_Customertypeid( 3 ); // C
    update( c );
end

rule "Clientes de tipo C asignar 300 como máximo de compras."
  no-loop true
  when
    c : SdtCustomer( gxTv_SdtCustomer_Customertypeid == 3 )
  then
    System.out.println("Clientes de tipo C asignar 300 como máximo de compras");
    c.setgxTv_SdtCustomer_Customermaxallowed( 300 );
end
```

Se realizan pruebas de concepto del lenguaje *DSL* que provee el motor de reglas para definir un lenguaje de reglas a un nivel de abstracción mayor y poder independizarse de los dialectos provistos por el motor. Se diseña un *DSL* para trabajar con el caso de uso según detalla la figura E.1.

Language Expression	Rule Language Mapping	Object	Scope
no hacer loop	no-loop true		[keyword]
There is an Instance with field of "{value}"	i: Instance(field=="{value}")		[condition]
Customer gender is "{gender}" and age is less than '{age}'	c : SdtCustomer(gxTv_SdtCustomer_...		[condition]
There is no current Instance with field : "{value}"	not Instance(field == "{value}")		[condition]
There is a Customer	c : SdtCustomer( )		[condition]
Customer age is greater than '{age}'	c : SdtCustomer(gxTv_SdtCustomer_...		[condition]
Customer gender between '{startAge}' and '{finishAge}'	c : SdtCustomer(gxTv_SdtCustomer...		[condition]
Customer Type is '{type}'	c : SdtCustomer(gxTv_SdtCustomer_...		[condition]
- Sin Id	cId:gxTv_SdtCustomer_CustomerId < 1		[condition]
- De Montevideo	gxTv_SdtCustomer_Countryid == 1,g...		[condition]
Log : "{message}"	System.out.println("{message}");		[consequence]
Set Customer Type to '{value}'	c.setgxTv_SdtCustomer_Customerty...		[consequence]
Create instance : "{value}"	insert(new Instance("{value}"));		[consequence]
Report error : "{error}"	System.err.println("{error}");		[consequence]
Retract the fact : '{variable}'	retract({variable}); //this would retrac...		[consequence]
Set Customer Maximum Allowed to '{value}'	java.math.BigDecimal a = new java....		[consequence]
Update the fact : '{variable}'	update({variable});		[consequence]

Figura E.1.: Objeto DSL de ejemplo en Drools

No es necesario definir las condiciones de las reglas con el lenguaje de expresiones MVEL [54] y el bloque de acciones con código Java ya que quedan escondidos en el objeto *DSL* (columna *Rule Language Mapping*). Un ejemplo definiendo reglas de negocio con el lenguaje de la figura E.1 se detalla a continuación, notar que se utilizan las expresiones de la columna *Language Expression*.

```
package com.sample
import com.sample.SdtCustomer;
expander Customer02.dsl

rule "Todos los clientes son B"
  no-loop true
  salience 100
  when
    There is a Customer
  then
    Log : "Todos los clientes son B"
    Set Customer Type to '2' // B
  end

rule "Clientes masculinos de hasta 20 años son de tipo A."
  no-loop true
  when
    Customer gender is "M" and age is less than '20'
```

```
  then
    Log : "Clientes masculinos de hasta 20 años son de tipo A"
    Set Customer Type to '1' // A
end

rule "Clientes femeninos de hasta 30 años son de tipo A."
  no-loop true
  when
    Customer gender is "F" and age is less than '30'
  then
    Log : "Clientes femeninos de hasta 30 años son de tipo A"
    Set Customer Type to '1' // A
end

rule "Clientes de mas de 70 años son de tipo C."
  no-loop true
  when
    Customer age is greater than '70'
  then
    Log : "Clientes de mas de 70 años son de tipo C"
    Set Customer Type to '3' // C
end

rule "Clientes entre 50 y 60 años asignar el tipo C."
  no-loop true
  when
    Customer gender between '50' and '60'
  then
    Log : "Clientes entre 50 y 60 años asignar el tipo C"
    Set Customer Type to '3' // C
    Update the fact : 'c'
end

rule "Clientes de tipo C asignar 300 como máximo de compras."
  no-loop true
  when
    Customer Type is '3'
  then
    Log : "Clientes de tipo C asignar 300 como máximo de compras"
    Set Customer Maximum Allowed to '300'
end
```

La ejecución fue satisfactoria. En este caso fue necesario crear un nuevo artefacto (de tipo *DSL*) en la plataforma para centralizar las definiciones del lenguaje.

Escenario 2 - Producto

Se implementan los dos juegos de reglas especificados en el escenario de uso de la sección G.2 utilizando el dialecto MVEL [54].

Se realizan pruebas utilizando tablas de decisión para definir las reglas de negocio;

es otra de las posibilidades que brinda el motor de reglas. La Figura E.2 detalla un ejemplo utilizando una tabla de decisión definida en un archivo Excel.

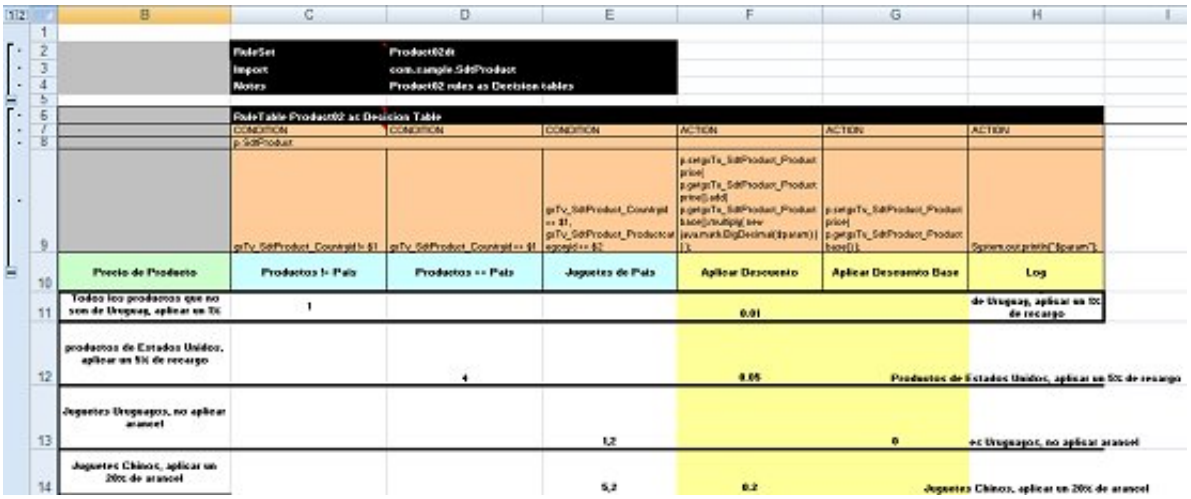


Figura E.2.: Tabla de decisión utilizando Drools

La ejecución fue satisfactoria.

Escenario 3 - Factura

Para resolver las reglas que plantea el escenario de la sección G.2, se escribió código Java extra para poder insertar en el motor los diferentes objetos sobre los cuales se definen reglas de negocio. No es suficiente con insertar el objeto `Invoice` con todo lo relacionado a la factura (Cliente, lista de líneas de Factura, Productos, etc).

A partir de la definición del Business Component `Invoice`, es necesario extraer y generar nuevos Business Components con información relacionada como `Customer` (el cliente que realizó la compra), `Product` (productos involucrados en la compra) e `Invoice.Line` (líneas de factura) de manera que el motor pueda "matchear" las diferentes reglas que evalúan condiciones sobre diferentes tipos de objetos.

En este escenario se realizan las siguientes tareas:

- Se implementa un juego de reglas básico para asegurarse que se detectan objetos de tipo `Invoice`, `Customer`, `Invoice.Line` y `Product`.
- Se implementan el juego de reglas detallado para el escenario.
  - Se investiga el recurso `agenda-group` provisto por el motor para agrupar reglas que conceptualmente aplican como un bloque, por ejemplo reglas relacionadas con validaciones, descuentos o promociones.
  - Se utiliza la posibilidad de enviar al motor de reglas variables que sean "globales" a la ejecución del motor.

Consideraciones

Para poder interactuar desde GeneXus con *Drools* es obligatorio calificar a las clases generadas con un *package name* de lo contrario el motor no funciona. Para el ejemplo se definió el *package name* `com.sample`.

Es necesario modificar la visibilidad de la variable `AnyError` (generada para todos los Business Components) a pública para que desde el motor de evaluación de reglas se pueda indicar que la ejecución de la parte derecha de una regla retornó error.

En el ejemplo de facturación (objeto `Invoice`) en particular cuando se manipulan las lineas de factura (`Items` de `Invoice`); se detectó que los ítems de una colección no tienen la referencia al Identificador del objeto padre, no se tiene forma de navegar desde una línea de factura a su cabezal.

Se realizan pruebas con *Drools Guvnor*[4] a los efectos de entender cómo funciona el BRMS.

Jena Rules

Se utiliza *Jena Rules* [41] con el modelo de inferencia predeterminado en donde se definen tanto las reglas, hechos como objetos de negocio utilizando *RDF*. Pruebas realizadas sobre Jena 2.5.7, en particular *Jena Rules*.

Para poder utilizar el mecanismo de inferencia es necesario transformar todos los objetos del dominio a RDF. En este caso es necesario transformar los objetos POJOs (Plain Old Java Objects) que se corresponden con el dominio a representación RPF; para ello se utiliza la biblioteca *jenabeans* [42] que permite transformar rápidamente un Objeto a formato RDF y viceversa.

Escenario 0 - Hello World

Se parte de la siguiente definición de reglas:

```
@prefix eg: http://com.sample/
# Hello World
[HelloWorld: (?m rdf:type eg:Message)
  (?m eg:message 'Hello World'^^xsd:string)
  (?m eg:status ?s)
  hide(eg:HelloWorld)
  noValue(?m eg:HelloWorld 'fired'
  →
  remove(1)
  remove(2)
  (?m eg:HelloWorld 'fired')
  (?m eg:status '1'^^xsd:int)
  (?m eg:message 'Goodbye cruel world'^^xsd:string)
```

[4] `http://www.jboss.org/drools/drools-guvnor.html`

```
  log(?m, 'HelloWold rule has matched', 'Changing Message status and message')
]

# Goodbye
[GoodBye: (?m rdf:type eg:Message)
  (?m eg:status '1'^^xsd:int)
  →
  log(?m, 'GoodBye rule has matched', 'do nothing')
]
```

la ejecución detalla los siguientes mensajes:

```
Log(HelloWorld) Node:http://com.sample/Message/17777129
  Reason:HelloWold rule has matched
  Action:Changing Message status and message
Log(GoodBye) Node:http://com.sample/Message/17777129
  Reason:GoodBye rule has matched
  Action:do nothing
```

Se tuvieron que realizar varias modificaciones a las reglas para poder lograr el efecto de encadenamiento de reglas al modificar el objeto `Mensaje` una vez que se ejecuta la regla `HelloWorld`:

- Para lograr modificar las propiedades de un objeto es necesario ingresarlo nuevamente al modelo *RDF*, es por eso que se utiliza la función `remove`.
- El agregar nuevamente el hecho con las modificaciones correspondientes, causa que se active nuevamente la regla `HelloWorld`.
- Se utiliza el recurso de agregar propiedades no visibles a la representación *RDF* y agregar una restricción por esta propiedad en las reglas de manera de asegurar que ciertas reglas no se activen nuevamente, en este caso en la regla `HelloWorld`, de lo contrario el motor quedaría en loop (verificar las funciones `hide` y `noValue` en el ejemplo anterior). Se utiliza el soporte de "*weak negation*" de Jena, expresado por el uso de la función `noValue`.
- Para lograr el efecto de Log, es decir enviar mensajes a salida estándar es necesario implementar clases que extiendan `BaseBuiltin`; y de esta manera se pueden agregar llamadas en las consecuencia de las reglas. Por más información ver la clase `MyLogger` que tiene un ejemplo de logging[5].

Integración GeneXus Java - Jena Rules

A partir de los fuentes GeneXus, se agrega un método que llama a una clase externa que realiza la llamada e interacción con el motor de reglas pasando como parámetro el correspondiente *Business Component*, es decir:

[5] `http://jena.sourceforge.net/inference/#RULEbuiltins`

```
mybr01.setCustomerBr01(customerbrinteraction01.this.AV8myCusto);
```

y el código de ejemplo que interactúa con el motor:

```
Model m = ModelFactory.createDefaultModel();
m.setNsPrefix("xsd", "http://www.w3.org/2001/XMLSchema#");
m.setNsPrefix("dc", "http://purl.org/dc/elements/1.1/");
m.setNsPrefix("owl", "http://www.w3.org/2002/07/owl#");
m.setNsPrefix("eg", "http://com.sample/");

myCustomer myCust = new myCustomer(myCustomer01.getStruct());

// Generate RDF Message (Using jenabeans)
Bean2RDF writer = new Bean2RDF(m);
writer.save(myCust);

Resource configuration = m.createResource();
configuration.addProperty(ReasonerVocabulary.PROPruleMode, "hybrid");
configuration.addProperty(ReasonerVocabulary.PROPruleSet, "Customer00.rules");
// Define myBuildInts
BuiltinRegistry.theRegistry.register(new MyLogger());

Reasoner reasoner = GenericRuleReasonerFactory.theInstance().create(configuration);
InfModel infmodel = ModelFactory.createInfModel(reasoner, m);
Validation(infmodel);
// force starting the rule execution
infmodel.prepare();
infmodel.write(System.out,"RDF/XML");
// write down the results in RDF/XML form
RDF2Bean reader = new RDF2Bean(m);
Collection<myCustomer> myCustomers = reader.load(myCustomer.class);
for (myCustomer myC : myCustomers)
{
  System.out.println(myC.getGxTv_SdtCustomer_Customerid() + " "
         + myC.getGxTv_SdtCustomer_Customername());
  // return values to original object
  myC.setValues(myCustomer01.getStruct());
}
```

Escenario 1 - Cliente

Se implementa un juego de reglas básico para asegurarse que se detectan objetos de tipo `Customer` (ver *Customer00.rules*).

Ejemplo de reglas:

```
@prefix eg: http://com.sample/
# Customer00 - Male Customer
[Customer00-Male_Customer:
  (?c rdf:type eg:myCustomer)
  (?c eg:gxTv_SdtCustomer_Customergender 'M'^^xsd:string)
```

```
  →
  log(?m, 'Customer is male', 'do nothing')
]

# Customer00 - Customer
[Customer00-Customer:
  (?m rdf:type eg:myCustomer)
  →
  log(?m, 'Customer', 'do nothing')
]

# Customer01 - Test
[Customer01-Test:
  (?m rdf:type rdfs:Class)
  →
  log(?m, 'startup', 'Rule Engine is running')
]

# Customer01 - Wrong Rule 1==2
[Customer01-Wrong_Rule:
  (?m rdf:type rdfs:Class)
  equal('1'^^xsd:int,'2'^^xsd:int)
  →
  log(?m, 'Wrong Rule', 'Wrong Rule')
]
```

Resultados obtenidos:

```
Log(Customer01-Test) Node:http://com.sample/myCustomer
  Reason:startup
  Action:Rule Engine is running
Log(Customer00-Customer) Node:http://com.sample/myCustomer/15794899
  Reason:Customer
  Action:do nothing
Log(Customer00-Male_Customer) Node:?m
  Reason:Customer is male
  Action:do nothing
```

Se implementan los dos juegos de reglas especificados en el escenario de uso de la sección G.2 (archivos *Customer01.rules* y *Customer02.rules*). El primer juego de reglas se pudo evaluar correctamente mientras que el segundo juego de reglas no fue satisfactorio.

Consideraciones

El orden de evaluación de reglas de la sección anterior no es el correcto. Se realizó una investigación y detectó que *Jena Rules* no tiene la posibilidad de declarar prioridades a las reglas; es decir lo que comúnmente se denomina `salience` en el contexto de los motores de evaluación de reglas.

El caso particular, se detectó al no ejecutarse la regla "*Clientes de tipo C asignar 300 como máximo de compras*".

Revisando documentación y foros se encontró la siguiente referencia:

> *The Jena rule engine was designed to make simple deductions, not transformations, and lacks the rich control features that fuller featured production systems have (such as rule salience)*[6].

Esto implica que queda descalificado *Jena Rules*, no se siguieron las pruebas con los siguientes casos de uso.

Jess

Se realizan pruebas básicas para el caso *Jess* [44] utilizando la versión 6.1p4 disponible con el libro "*Jess In Action*" [151].

Escenario 0 - Hello World

Funcionamiento correcto según lo especificado para el siguiente juego de reglas (ver *JessTest.java* y *Sample.jess*):

```
;;Imports
(import com.sample.*)
; Templates
(defclass message com.sample.Message)

; Hello World
(defrule Hello_World
 (declare (salience -100))
 ?msg ←
 (message
  (message "Hello World") (status ?actualstatus) (OBJECT ?obj))
 ⇒
 (printout t "Hello_World" crlf)
 (modify ?msg ( message "Goodbye cruel world"))
 (set ?obj status ( + ?actualstatus 1))
 (printout t "Changing " ?msg ": status=" (get ?obj status)
             " message=" (get ?obj message) crlf)
 (update ?obj)
)

;Goodbye
(defrule GoodBye
 (message (message ?m) (status 1) (OBJECT ?obj) )
 ⇒
 (printout t "GoodBye" crlf)
```

[6] http://eprints.otago.ac.nz/164/1/dp2005-12.pdf

```
 (printout t "Checking Bean status=" (get ?obj status) crlf)
)
```

se detallan los siguientes mensajes al ejecutar el caso de uso:

```
Hello_World
Changing <Fact-1>: status=1 message=Goodbye cruel world
GoodBye
Checking Bean status=1
```

Integración GeneXus Java - Jess

A partir de los fuentes *Java* GeneXus, se agrega un método que llama a una clase externa que realiza la llamada e interacción con el motor de reglas pasando como parámetro el correspondiente *Business Component*:

```
mybr01.setCustomerBr01(customerbrinteraction01.this.AV8myCusto);
```

y el código de ejemplo que interactúa con el Shell Jess es el siguiente:

```
public static void setCustomerBr01( SdtCustomer myCustomer01 ) {
  try {
    String ruleBase = "C:/java/.../com/sample/Customer02.jess";
    //load up the Rulebase
    Rete r = readRule(ruleBase);

    myCustomer myCust = new myCustomer(myCustomer01.getStruct());
    insertInWM(myCust, r );
    fireRules(r);
    myCust.setValues(myCustomer01.getStruct());
  }
  catch (Throwable t) {
    t.printStackTrace();
  }
}

private static void insertInWM(Object myObject,Rete r) throws JessException
{
  String functionCall = "definstance";
  String name = "customer";

  Funcall f = new Funcall(functionCall, r);
  f.add(new Value(name, RU.ATOM));
  f.add(new Value(myObject));
  Value v = f.execute(r.getGlobalContext())
  String s = v.toString();
}
```

```
private static Rete readRule(String string) throws JessException {
  String batch = "(batch \"" + string + "\")";
  Rete r = new Rete();
  r.executeCommand(batch);
  r.executeCommand("(reset)");
  return r;
}

private static void fireRules(Rete r) throws JessException {
  r.run();
}
```

Escenario 1 - Cliente

Se implementa un juego de reglas básico para asegurarse que se detectan objetos de tipo `Customer` (*Customer00.jess*):

```
;;Imports
(import com.sample.*)
;Templates
(defclass customer com.sample.myCustomer )
; Configuration
(watch all )
(agenda)
(get-strategy)
(jess-version-string)

; Male Customer
(defrule Male_Customer
 ?c ← (customer
  (gxTv_SdtCustomer_Customergender "M") (OBJECT ?obj))
 ⇒
 (printout t "Customer is male" crlf)
)

; Customer
(defrule Customer
 ?c ← (customer (OBJECT ?obj))
 ⇒
 (printout t "Customer" crlf)
)
```

Ejemplo de resultados obtenidos:

```
Customer is male
Customer
```

Se implementan los dos juegos de reglas especificados en el caso de uso de la sección G.2 (ver las reglas en los archivos *Customer01.jess* y *Customer02.jess*):

```
;;Imports
(import com.sample.*)
; Templates
(defclass customer com.sample.myCustomer )

;; Rules
(defrule startup (test (eq 1 1))
 ⇒
 (printout t "Rule engine is running" crlf)
)

; Todos los clientes son B
(defrule Todos_los_clientes_son_B
 ?c ← (customer (OBJECT ?obj))
 ⇒
 (printout t "Todos_los_clientes_son_B" crlf)
 (set ?obj gxTv_SdtCustomer_Customertypeid 2 )
)

; Clientes masculinos de hasta 25 años son de tipo A
(defrule Clientes_masculinos_de_hasta_25_anos_son_de_tipo_A
 ?c ← (customer
  (gxTv_SdtCustomer_Customergender "M")
  (gxTv_SdtCustomer_Customerage ?age ) (OBJECT ?obj) )
  (test (<= ?age 25))
 ⇒
 (printout t "Clientes_masculinos_de_hasta_25_anos_son_de_tipo_A" crlf)
 (set ?obj gxTv_SdtCustomer_Customertypeid 1 )
)

; Clientes femeninos de hasta 30 años son de tipo A
(defrule Clientes_femeninos_de_hasta_30_anos_son_de_tipo_A
 ?c ← (customer
  (gxTv_SdtCustomer_Customergender "F")
  (gxTv_SdtCustomer_Customerage ?age ) (OBJECT ?obj) )
  (test (<= ?age 30))
 ⇒
 (printout t "Clientes_femeninos_de_hasta_30_anos_son_de_tipo_A" crlf)
 (set ?obj gxTv_SdtCustomer_Customertypeid 1 )
)

; Clientes de mas de 70 años son de tipo C
(defrule Clientes_de_mas_de_70_anos_son_de_tipo_C
 ?c ← (customer
  (gxTv_SdtCustomer_Customerage ?age ) (OBJECT ?obj) )
  (test (> ?age 70))
 ⇒
 (printout t "Clientes de mas de 70 años son de tipo C" crlf)
 (set ?obj gxTv_SdtCustomer_Customertypeid 3 )
)
```

Segundo juego de reglas:

```
;;Imports
(import com.sample.*)
; Templates
(defclass customer com.sample.myCustomer )
;; Configuration;
(watch all )

; Todos los clientes son B
(defrule Todos_los_clientes_son_B
 ?c ← (customer (OBJECT ?obj))
 ⇒
 (printout t "Todos_los_clientes_son_B" crlf)
 (set ?obj gxTv_SdtCustomer_Customertypeid 2)
)

; Clientes masculinos de hasta 20 años son de tipo A
(defrule Clientes_masculinos_de_hasta_20_anos_son_de_tipo_A
 ?c ← (customer
  (gxTv_SdtCustomer_Customergender "M")
  (gxTv_SdtCustomer_Customerage ?age)(OBJECT ?obj))
  (test (<= ?age 20 ))
 ⇒
 (printout t "Clientes_masculinos_de_hasta_20_anos_son_de_tipo_A" crlf)
 (set ?obj gxTv_SdtCustomer_Customertypeid 1)
 (update ?obj)
)

; Clientes femeninos de hasta 30 años son de tipo A
(defrule Clientes_femeninos_de_hasta_30_anos_son_de_tipo_A
 ?c ← (customer
  (gxTv_SdtCustomer_Customergender "F")
  (gxTv_SdtCustomer_Customerage ?age)(OBJECT ?obj))
  (test (<= ?age 30 ))
 ⇒
 (printout t "Clientes_femeninos_de_hasta_30_anos_son_de_tipo_A" crlf)
 (set ?obj gxTv_SdtCustomer_Customertypeid 1)
)

; Clientes de mas de 70 años son de tipo C
(defrule Clientes_de_mas_de_70_anos_son_de_tipo_C
 ?c ← (customer
  (gxTv_SdtCustomer_Customerage ?age ) (OBJECT ?obj) )
  (test (> ?age 70 ))
 ⇒
 (printout t "Clientes_de_mas_de_70_anos_son_de_tipo_C" crlf)
 (set ?obj gxTv_SdtCustomer_Customertypeid 3 )(update ?obj)
)

; Clientes entre 50 y 60 años asignar el tipo C
(defrule Clientes_entre_50_y_60_anos_asignar_el_tipo_C
 ;(declare (no-loop TRUE))
 ?c ← (customer
```

```
  (gxTv_SdtCustomer_Customerage ?age ) (OBJECT ?obj) )
  (test (>= ?age 50 ))
  (test (<= ?age 60 ))
 ⇒
 (printout t "Clientes_entre_50_y_60_anos_asignar_el_tipo_C" crlf)
 (set ?obj gxTv_SdtCustomer_Customertypeid 3 )
 (modify ?c (gxTv_SdtCustomer_Customertypeid 3))
)

; Clientes de tipo C asignar 300 como máximo de compras
(defrule Clientes_de_tipo_C_asignar_300_como_maximo_de_compras
 ?c ← (customer
  (gxTv_SdtCustomer_Customertypeid 3) (OBJECT ?obj) )
 ⇒
 (printout t "Clientes_de_tipo_C_asignar_300_como_maximo_de_compras" crlf)
 (set ?obj gxTv_SdtCustomer_Customermaxallowed 300 )
 (update ?obj)
)
```

Ejemplo de resultados obtenidos:

```
Todos_los_clientes_son_B
Rule engine is running
Insert OK
```

Consideraciones

Los objetos que se insertan en la memoria de trabajo del motor de evaluación *Jess* tienen que satisfacer la especificación JavaBeans [38]. Para poder interactuar con los objetos *Java* dentro del Shell *Jess* es necesario que los objetos acepten la especificación *PropertyChangeListeners*[7].

Por defecto las clases estándar generadas por GeneXus (*SdtCustomer* y *StructSdtCustomer* del ejemplo) no funcionaron debido a esto. Se crea una clase con todas las propiedades que tiene el *Business Component* y además el soporte de *PropertyChangeListeners*. *Jess* interactúa con estos objetos y posteriormente se pasan todas las propiedades al objeto original para que siga la ejecución normal del programa.

El último juego de reglas del caso *Customer02.jess*, en las consecuencias de las reglas se actualiza el mismo objeto; el motor queda en loop debido a que se activa y ejecuta nuevamente la misma regla. Si bien existe la cláusula `(declare (no-loop TRUE))` para evitar éste comportamiento, se encuentra disponible en una versión del motor mas nueva que la que se probó (versión 7.0[8]).

[7] `http://java.sun.com/j2se/1.5.0/docs/api/java/beans/package-summary.html`
[8] `http://www.jessrules.com/docs/70/release_notes.html`

En términos generales para poder utilizar el motor *Jess* integrado a GeneXus sería necesario hacer modificaciones al generador GeneXus *Java* para que agregue más código para poder utilizar el recurso de listeners que *Jess* necesita para poder actualizar las propiedades de objetos una vez que el contexto de ejecucion cambia al intérprete *Jess*.

Teniendo en cuenta el punto anterior y que *Jess* necesita una licencia comercial, no se siguieron las pruebas de los escenarios *Producto* y *Factura*.

NxBRE

Pruebas realizadas utilizando NxBRE 3.2.0 [55]. El motor dispone de dos mecanismos de ejecución de reglas:

- Flujo de reglas (RuleFlow).
- Motor de inferencia (InferenceEngine).

Escenario 0 - Hello World

Ejemplo de especificación de regla utilizando la sintaxis propietaria de NxBRE:

```
<?xml version="1.0" encoding="UTF-8"?>
<xBusinessRules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:noNamespaceSchemaLocation="xBusinessRules.xsd">
  <!- global values ->
  <Short id="myStatusDummy01" value="1"/>
  <String id="myHelloWorldMessageDummy01" value="Hello World"/>
    <Logic>
    <!- Message( status == Msg.HELLO ) ->
    <If>
      <And>
      <Equals leftId="myMessageComparer" rightId="myHelloWorldMessage">
        <ObjectLookup id="myMessageComparer" objectId="myMessage"
                      member="Message"/>
        <String id="myHelloWorldMessage" value="Hello World"/>
      </Equals>
    </And>
    <Do>
      <!- change Variable value ->
      <Modify id="myHelloWorldMessage" value="Good bye cruel world"/>
      <!-log information->
      <Log level="3" msg="Hello World triggered"/>
      <!- Change object values ->
      <ObjectLookup id="myMessage2" member="setValues" objectId="myMessage"
                    type="nxbre01.Msg">
        <Argument type="String" valueId="myHelloWorldMessage"/>
        <Argument type="Short" value="1"/>
      </ObjectLookup>
```

```
      <!-Remove object from working memory->
      <Retract id="myMessage"/>
      <!-Reassing object to force the Rule2 triggering ->
      <Assert id="myMessage" type="nxbre01.Msg,nxbre01">
        <Argument type="String" valueId="myHelloWorldMessage"/>
        <Argument type="Short" value="1"/>
      </Assert>
    </Do>
    </If>
      <Else>
        <Logic>
        <If>
          <And>
            <Equals leftId="myStatusComparer" rightId="myStatus">
              <ObjectLookup id="myStatusComparer" objectId="myMessage"
                            member="Status"/>
              <Short id="myStatus" value="1"/>
          </Equals>
        </And>
    <Do>
      <Log level="3" msg="Goodbye cruel world triggered"/>
    </Do>
  </If>
  </Logic>
  </Else>
  </Logic>
</xBusinessRules>
```

al ejecutarse las reglas el resultado esperado es el siguiente:

```
Hello World triggered
Goodbye cruel world triggered
```

Sin embargo el resultado obtenido es:

```
Hello World triggered
```

Consideraciones

La evaluación del escenario anterior no fue satisfactoria. Se revisa la documentación del modo de ejecución *RuleFlow* y encuentra que el motor no es *forward chaining*; por lo que no se dispara la segunda regla al modificarse el objeto en el escenario de la sección G.2. Esta forma de ejecución queda automáticamente descalificada y se realiza una prueba de concepto utilizando el motor de inferencia. No se logra que las reglas en formato RuleML se ejecuten, por lo que no se sigue con las demás pruebas.

Drools.Net

Pruebas realizadas sobre el port 3.0 de Drools.NET [14].

Escenario 0 - Hello World

Funcionamiento correcto según lo especificado para el siguiente juego de reglas:

```
package drools01
import drools01;

rule "Hello World"
  when
    m : Msg( Status == 0 , myMessage : Message )
  then
    System.Console.WriteLine(myMessage);
    m.Message = "Goodbye cruel world";
    m.Status = Msg.GOODBYE;
    modify( m );
end

rule "GoodBye"
  when
    m : Msg( Status == 1 , myMessage : Message )
  then
    System.Console.WriteLine(myMessage);
end

rule "Bootstrap"
  no-loop true
  salience 10
  when
    // nothing
  then
    System.Console.WriteLine("Rule Engine is running");
end
```

se detallan los siguientes mensajes:

```
Rule Engine is running
Hello World
Goodbye cruel world
```

Integración GeneXus C# - Drools.Net

A partir de los fuentes C# GeneXus, se agrega un método que llama a una clase externa que realiza la llamada e interacción con el motor de reglas pasando como parámetro el correspondiente *Business Component*:

```
mybr01.setInvoiceBr01( this.AV9myInvoi ); // Manually added
```

y el código de ejemplo que interactúa con el motor:

```
public static void setCustomerBr01( SdtCustomer myCustomer01) {
 try {
  Stream stream = null;
  String aXMLFile = @"C:\...\rules\Customer02.drl";
  stream = new StreamReader(aXMLFile).BaseStream;
  PackageBuilder builder = new PackageBuilder();
  builder.AddPackageFromDrl(stream);
  Package pkg = builder.GetPackage();
  RuleBase ruleBase = RuleBaseFactory.NewRuleBase();
  ruleBase.AddPackage(pkg);
  WorkingMemory workingMemory = ruleBase.NewWorkingMemory();
  insertInWM(myCustomer01, workingMemory);
  workingMemory.fireAllRules();
  workingMemory.dispose();}
 catch (Exception e){
  Console.WriteLine(e.Message);
  Console.WriteLine(e.StackTrace);}
}

private static void insertInWM(Object myObject, WorkingMemory wm) {
  bool dynamic = false;
  // do not use beans
  wm.assertObject(myObject, dynamic);
}
```

Escenario 1 - Cliente

Se implementa un juego de reglas básico para asegurarse que se detectan objetos de tipo *Customer* (ver *Customer00.rules*).

Ejemplo de reglas:

```
package GeneXus.Programs
import GeneXus.Programs;

rule "Male Customer"
  when
    c : SdtCustomer( gxTpr_Customergender == "M")
  then
    System.Console.WriteLine("Customer is male");
end

rule "Customer"
  salience 10
  when
    m : SdtCustomer( )
  then
    System.Console.WriteLine("Customer");
end

rule "Bootstrap"
```

```
  no-loop true
  salience 10
  when
    // nothing
  then
    System.Console.WriteLine("Rule Engine is running");
end
```

Resultados obtenidos:

```
Customer
Rule Engine is running
Customer is male
```

Se implementan los dos juegos de reglas especificados en el escenario de uso de la sección G.2 (ver las clases *Customer01.drl* y *Customer02.drl*):

```
package GeneXus.Programs
import GeneXus.Programs;

rule "Todos los clientes son B"
  no-loop true
  when
    c : SdtCustomer( )
  then
    System.Console.WriteLine("Todos los clientes son B");
    c.gxTpr_Customertypeid = 2; // B
    modify( c );
end

rule "Clientes masculinos de hasta 25 años son de tipo A."
  no-loop true
  when
    c : SdtCustomer( gxTpr_Customergender == "M" ,
                     gxTpr_Customerage <= 25 )
  then
    System.Console.WriteLine("Clientes masculinos de hasta 25 años son de
tipo A");
    c.gxTpr_Customertypeid = 1; // A
end

rule "Clientes femeninos de hasta 30 años son de tipo A."
  no-loop true
  when
    c : SdtCustomer( gxTpr_Customergender == "F" ,
                     gxTpr_Customerage <= 30 )
  then
    System.Console.WriteLine("Clientes femeninos de hasta 30 años son de
tipo A");
    c.gxTpr_Customertypeid = 1; // A
end
```

```
rule "Clientes de mas de 70 años son de tipo C."
  no-loop true
  when
    c : SdtCustomer( gxTpr_Customerage > 70 )
  then
    System.Console.WriteLine("Clientes de mas de 70 años son de tipo C");
    c.gxTpr_Customertypeid = 3; // C
end
```

Segundo juego de reglas:

```
package GeneXus.Programs
import GeneXus.Programs;

rule "Todos los clientes son B"
  no-loop true
  salience 100 // aumento prioridad
  when
    c : SdtCustomer( )
  then
    System.Console.WriteLine("Todos los clientes son B");
    c.gxTpr_Customertypeid = 2; // B
end

rule "Clientes masculinos de hasta 20 años son de tipo A."
  no-loop true
  when
    c : SdtCustomer( gxTpr_Customergender == "M" ,
                     gxTpr_Customerage <= 20 )
  then
    System.Console.WriteLine("Clientes masculinos de hasta 20 años son de
tipo A");
    c.gxTpr_Customertypeid = 1; // A
end

rule "Clientes femeninos de hasta 30 años son de tipo A."
  no-loop true
  when
    c : SdtCustomer( gxTpr_Customergender == "F" ,
                     gxTpr_Customerage <= 30 )
  then
    System.Console.WriteLine("Clientes femeninos de hasta 30 años son de
tipo A");
    c.gxTpr_Customertypeid = 1; // A
end

rule "Clientes de mas de 70 años son de tipo C."
  no-loop true
  when
    c : SdtCustomer( gxTpr_Customerage > 70 )
```

```
  then
    System.Console.WriteLine("Clientes de mas de 70 años son de tipo C");
    c.gxTpr_Customertypeid = 3; // C
end

rule "Clientes entre 50 y 60 años asignar el tipo C."
  no-loop true
  when
    c : SdtCustomer( gxTpr_Customerage >= 50 ,
                     gxTpr_Customerage <= 60 )
  then
    System.Console.WriteLine("Clientes entre 50 y 60 años asignar el tipo
C");
    c.gxTpr_Customertypeid = 3; // C
    modify( c );
end

rule "Clientes de tipo C asignar 300 como máximo de compras."
  no-loop true
  when
    c : SdtCustomer( gxTpr_Customertypeid == 3 )
  then
    System.Console.WriteLine("Clientes de tipo C asignar 300 como máximo
de compras");
    c.gxTpr_Customermaxallowed = 300;
end
```

Resultados obtenidos:

```
Todos los clientes son B
Clientes femeninos de hasta 30 años son de tipo A
Insert OK
```

La ejecución fue satisfactoria.

Escenario 2 - Producto

No se realiza la implementación de este escenario debido a que es similar al anterior.

Escenario 3 - Factura

Al igual que el caso Java con *Drools*, para resolver las reglas planteadas en éste escenario se tuvo que escribir código C# para poder insertar en el motor los diferentes objetos sobre los cuales se definen reglas de negocio.

Consideraciones

Se utilizaron los archivos *drl* (Drools Rules Language) de la implementación Java realizando algunas modificaciones para ajustar la generación de código C#.

- *Package*: se ajusta al package predeterminado GeneXus; `package GeneXus.Programs`.
- *Imports*: se importa todo el namespace GeneXus, utilizando la siguiente palabra clave: `import GeneXus.Programs`.
- *Getter* y *Setters*: se modifican las referencias a *getters* y *setters* Java por propiedades de .Net según generación C# GeneXus.

Al utilizar el port 3.0 algunas funcionalidades utilizadas en la versión Java no se soportan:

- No se pudo referenciar variables estáticas de la clase al comparar con condiciones de reglas.
- No se soporta la sintaxis `$i` para identificar proyecciones en el "matcheo" de patrones.
- Para hacer comparaciones entre propiedades de diferentes objetos es necesario proyectar primero en variables y después realizar la comparación. Es decir, en vez de utilizar el siguiente código válido para la versión Java:

```
c : SdtCustomer( )
i : SdtInvoice(
    c.getgxTv_SdtCustomer_Customerid == gxTv_SdtInvoice_Customerid
    )
```

  se debe comparar de la siguiente forma:

```
c : SdtCustomer( cutomerId : gxTpr_Customerid )
i : SdtInvoice( gxTpr_Customerid == cutomerId )
```

Al igual que el caso Java, es necesario modificar la visibilidad de la variable `AnyError` a pública para que el motor de reglas pueda indicar que la manipulación de un *Business Component* retornó error.

Ruleby

Ruleby [85] es una gema *Ruby* que implementa el algoritmo *Rete*. Se realizan pruebas básicas utilizando la versión 0.4.

Escenario 0 - Hello World

Funcionamiento correcto según lo especificado para el siguiente juego de reglas:

```
class HelloWorldRulebook < Rulebook
  def rules
    rule [Message, :m, m.status == :HELLO] do |v|
      puts v[:m].message
      v[:m].message = "Goodbye world"
```

```
      v[:m].status = :GOODBYE
      modify v[:m]
    end

    rule [Message, :m, m.status == :GOODBYE] do |v|
      puts v[:m].message
    end
  end
end
```

se detallan los siguientes mensajes:

```
Hello World
Goodbye world
```

Integración GeneXus Ruby - Ruleby

A partir de los fuentes Ruby GeneXus, se agrega un método que llama a una clase externa que realiza la llamada e interacción con el motor de reglas pasando como parámetro el correspondiente *Business Component*:

```
@dummy01 = Brsample01::Mybr01.new()
@dummy01.setcustomerbr01(@av8mycusto)
```

y el código de ejemplo que interactúa con la gema es:

```
public
  def setcustomerbr01( av8mycusto )
    @av8mycusto = av8mycusto
    engine :engine do |e|
      CustomerRules00.new(e).rules
      e.assert @av8mycusto
      e.match
    end
  end
```

Escenario 1 - Cliente

Se implementa un juego de reglas básico para asegurarse que se detectan objetos de tipo `Customer`, ver la clase *Customer00* en el archivo *mybr01.rb* que implementa el DSL interno de representación de reglas.

```
class CustomerRules00 < Rulebook
  def rules
    rule :Customer,
    [SdtCustomer, :c,] do |v|
      puts "Customer"
```

```
    end

    rule :Male_Customer,
    [SdtCustomer, :c, m.customergender == 'M'] do |v|
      puts "Customer is Male"
    end
  end
end # Customer00 class
```

Resultados obtenidos:

```
Customer is Male
Customer
```

Se implementan los dos juegos de reglas especificados en el escenario de la sección G.2; las clases *CustomerRules01* y *CustomerRules02* heredan de la clase *Rulebook* la cual especifica el DSL interno "rules" donde se deben de especificar las reglas.

```
class CustomerRules01 < Rulebook
  def rules
    rule :Todos_los_clientes_son_B,
    [SdtCustomer, :c] do |v|
      puts "Todos los clientes son B"
      v[:c].customertypeid = "2"
    end

    rule :Clientes_masculinos_de_hasta_25_años_son_de_tipo_A,
    [SdtCustomer, :c, m.customergender == 'M',
                      m.customerage <= 25]
    do |v|
      puts "Clientes masculinos de hasta 25 años son de tipo A"
      v[:c].customertypeid = "1"
    end

    rule :Clientes_femeninos_de_hasta_30_años_son_de_tipo_A,
    [SdtCustomer, :c, m.customergender == 'F',
                      m.customerage <= 30]
    do |v|
      puts "Clientes femeninos de hasta 30 años son de tipo A"
      v[:c].customertypeid = "1"
    end

    rule :Clientes_de_mas_de_70_años_son_de_tipo_C,
    [SdtCustomer, :c, m.customerage > 70] do |v|
      puts "Clientes de mas de 70 años son de tipo C"
      v[:c].customertypeid = "3"
    end
  end
end # Customer01 class
```

Segundo juego de reglas:

```
class CustomerRules02 < Rulebook
  def rules
    rule :Todos_los_clientes_son_B,
      {:priority => 100}, # aumento prioridad
    [SdtCustomer, :c] do |v|
      puts "Todos los clientes son B"
      v[:c].customertypeid = "2"
    end

    rule :Clientes_masculinos_de_hasta_20_años_son_de_tipo_A,
    [SdtCustomer, :c, m.customergender == 'M',
                      m.customerage <= 20]
    do |v|
      puts "Clientes masculinos de hasta 20 años son de tipo A"
      v[:c].customertypeid = "1"
    end

    rule :Clientes_femeninos_de_hasta_30_años_son_de_tipo_A,
    [SdtCustomer, :c, m.customergender == 'F',
                      m.customerage <= 30]
    do |v|
      puts "Clientes femeninos de hasta 30 años son de tipo A"
      v[:c].customertypeid = "1"
    end

    rule :Clientes_de_mas_de_70_años_son_de_tipo_C,
    [SdtCustomer, :c, m.customerage > 70] do |v|
      puts "Clientes de mas de 70 años son de tipo C"
      v[:c].customertypeid = "3"
    end

    rule :Clientes_entre_50_y_60_años_asignar_el_tipo_C,
    [SdtCustomer, :c, m.customerage >= 50,
                      m.customerage <= 60]
    do |v|
      puts "Clientes entre 50 y 60 años asignar el tipo C"
      v[:c].customertypeid = "3"
    end

    rule :Clientes_de_tipo_C_asignar_300_como_máximo_de_compras,
    [SdtCustomer, :c, m.customertypeid == 3] do |v|
      puts "Clientes de tipo C asignar 300 como máximo de compras"
      v[:c].customermaxallowed = "300"
    end
  end
end # Customer02 class
```

Resultados obtenidos:

```
Todos los clientes son B
Clientes femeninos de hasta 30 años son de tipo A
```

```
Data has been successfuly added.
Insert OK
```

El funcionamiento del juego de reglas fue satisfactorio.

Consideraciones

Hay que tener en cuenta que la gema implementa las características básicas de *Rete* detectándose lo siguiente:

- Si en las consecuencias de las reglas se actualiza el mismo objeto (por ejemplo se realiza un `update`) que se encuentra referenciado en las condiciones de la misma regla el motor queda en loop debido a que se activa y ejecuta nuevamente.
- No existen propiedades del estilo `no-loop` para evitar este tipo de comportamiento. Se debería implementar un mecanismo para que las reglas no se activen nuevamente. Se podría utilizar la misma solución que se encontró en el caso *Jena Rules*; donde se agregan nuevas propiedades a los objetos que se utilizan como bandera para indicar que una regla ya fue ejecutada.

No se continuaron las pruebas de los escenarios relacionados *Producto* y *Factura* del Anexo G.

E.3. Selección

Para la plataforma Java se evaluaron las herramientas *Jess*, *JenaRules* y *Drools*.

Jena Rules, si bien es un framework específico Web Semántica, posee motores de inferencia que utiliza algoritmos de tipo forward y backward chaining. Se realiza una prueba de concepto con algunos casos de uso básicos no calificando para los propósitos del proyecto debido a algunas restricciones encontradas como por ejemplo la inexistencia de mecanismos que permitan declarar prioridades entre reglas.

El caso *Jess* es un motor de reglas que hace tiempo existe en el mercado, si bien no es open-source es muy utilizado en el ambiente académico debido a que no tiene licencia en este caso. Las pruebas realizadas fueron parcialmente satisfactorias ya que no se logró implementar en su totalidad los escenarios planteados.

Para la plataforma Java, *Drools* parece ser el motor de reglas más completo. Se completaron en forma satisfactoria todos los escenarios de evaluación e investigaron mecanismos de definición de reglas en tablas de decisión y lenguajes específicos de dominio. Tiene una fuerte comunidad utilizándolo además de estar respaldado por *JBoss* y *RedHat*.

En la plataforma .Net se evaluaron *NXBre* y *Drools.Net*, no teniendo en cuenta las propuestas comerciales.

Para el caso *NXBre* no se pudo completar los escenarios de evaluación propuestos, no funcionaron los ejemplos de definición de reglas en lenguaje *RuleML*.

Para el caso *Drools.Net* si bien es un port de *Drools 3.0*, se ejecutaron los casos de uso propuestos.

Para *Ruby* sólo se hicieron pruebas con la gema *RuleBy* ya que fue la única herramienta que calificó la pre-selección. No se implementaron todos los escenarios ya que se encontró una restricción (que es posible solucionar) cuando se definen reglas que condicionan y a la vez actualizan el mismo objeto.

La selección de herramientas para las plataformas de ejecución GeneXus según la evaluación realizada son: *Drools* para Java, *Drools.Net* para .Net y *Ruleby* para Ruby.

F. Gramática y Lenguajes

El problema de la definición de un lenguaje sobre el cual se puedan manipular reglas de negocio para la plataforma GeneXus se ataca en dos niveles. Por un lado se crea un lenguaje técnico de reglas denominado *GXRL - GeneXus Rule Language* al mismo nivel de abstracción que el lenguaje GeneXus.

Por otro lado, con el objetivo de brindar un mecanismo en donde el analista de negocio pueda entender las reglas que se aplican en los contextos de ejecución, se desarrolla un DSL denominado *GXRDSL - GeneXus Rule DSL*. Se busca posibilitar la personalización del lenguaje de reglas para poder interactuar con el experto del negocio.

F.1. GeneXus Rule Language

Para especificar la gramática del lenguaje se utiliza notación BNF. Los nodos no terminales se representan entre "<" y ">", los nodos terminales en negrita, "::=" define la composición de los nodos no terminales, "|" se utiliza para elegir entre diferentes alternativas.

Se utiliza la herramienta Gold Parser (Grammar Oriented Language Developer) 3.4.4 [27] para definir dicha gramática.

Gold Parser

GOLD es el acrónimo de **G**rammar **O**riented **L**anguage **D**eveloper, es un parser de gramáticas BNF (Backus-Naur Form).

La aplicación analiza la gramática y salva la tabla de parsing en un archivo separado. Dicho archivo puede ser cargado posteriormente por el motor de parsing. La tabla de parsing es independiente al lenguaje de programación, por lo que el motor de parsing puede implementarse fácilmente en diferentes lenguajes.

Algunas características:

- Implementa un DFA (Deterministic Finite Automata); un tipo de grafo para reconocer patrones para implementar el administrador de tokens. Es determinístico en el sentido de que a partir de cualquier nodo, solo puede existir un camino para una entrada específica. Es decir, no existe ambigüedad al moverse entre los nodos.

- El grafo es finito, existe un número fijo y conocido de nodos (estados) y ramas.
- El grafo es un autómata; las transiciones entre estados está completamente determinada por su entrada. El algoritmo simplemente se encarga de seguir la rama correcta basada en la entrada leída.
- Cada vez que se identifica un toquen, es pasado al motor de parsing LALR (Lookahead Left to Right Parsing) y el tokenizer (administrador de tokens) reinicializa su estado.
- Los terminales se representan mediante expresiones regulares.
- Los juegos de caracteres se basan en notación de conjuntos (set notation).

Metadata de la Gramática

La metadata de la gramática consta de las siguientes propiedades.

```
"Name" = 'Ruleset basic grammar'
"Author"= 'Luciano Silveira'
"Version"= '0.2'
"About"= 'Grammar to work with GeneXus Rulesets'
"Case Sensitive"= False
"Start Symbol" = <Statements>
"Character Mapping"= Unicode
```

A partir de la gramática definida se utiliza el motor C# Morozov [26] para crear los fuentes básicos C# de parsing.

A continuación se especifican los componentes principales asociados con el lenguaje técnico.

Sets

La notación de grupos es la siguiente:

```
{ID Head} = {Letter}+[_]
{ID Tail} = {Alphanumeric}+[_]
{String Chars} = {Printable}+{HT}-["]
{String Extended Chars} = {Printable Extended}+{HT}-["]
{Char Ch} = {Printable}+{HT}-['']
{Char Extended Ch} = {Printable}+{HT}-['']
{Hex Digit} = {Digit}+[abcdef]+[ABCDEF]
{WS} = {Whitespace}-{CR}-{LF}
{Start Name Char} = {Letter}
```

Comentarios

Los comentarios son secciones de texto que se ignoran por completo, se eliminan en el proceso de parsing.

Se permiten comentarios por bloque utilizando los caracteres `/* */` y comentar una línea de código utilizando el carácter `//`.

```
Comment Line = '//'
Comment Start = '/*'
Comment End = '*/'
```

Terminales

Expresiones regulares que definen los nodos terminales.

```
String = ''({String Chars} | {String Extended Chars})*''
    | '"' ({String Chars} | {String Extended Chars})* '"'
Int = ('-')?{Digit}+
Float = {Digit}*'.'{Digit}+
Bool = ('true' | 'false')
Then = 'then'
When = 'when'
Name = {Start Name Char}{Id Tail}*
Equal = '=='
NotEqual = '<>'
Assign = '='
Update = 'update'
Modify = 'modify'
Retract = 'retract'
Insert = 'insert'
InsertLogical = 'insertLogical'
Rule = 'rule'
FunctionCommandName = 'msg'
Ampersand = '&'
End = 'end'
And = ('and' | '.and.')
Or = ('or' | '.or.')
RuleEngineReference = 'RuleEngine'
NewOperator = 'new'
```

Palabras clave

Lista de palabras clave:

```
true               |   false          |   null
then               |   when           |   lock-on-active
date-effective     |   no-loop        |   auto-focus
activation-group   |   agenda-group   |   salience
enabled            |   rule           |   eval
not                |   or             |   and
exists             |   end
```

No Terminales

Expresiones BNF de nodos no terminales.

Regla

Un archivo de reglas es una lista de reglas:

```
<Statements>
    ::= <RuleStatement>
    |   <Statements> <RuleStatement>
<RuleStatement>
    ::= <Rule>
<RuleAttributes>
    ::= <RuleAttributes> <RuleAttribute>
    |   ! Nothing
<RuleAttribute>
    ::= <Salience>
    |   <NoLoop>
    |   <AgendaGroup>
    |   <Duration>
    |   <ActivationGroup>
    |   <AutoFocus>
    |   <DateEffective>
    |   <Enabled>
    |   <LockOnActive>
<Rule>
    ::= Rule <RuleId> <RhsChunk> End
    |   Rule <RuleId> <RuleAttributes> <RhsChunk> End
    |   Rule <RuleId> <RuleAttributes> <WhenPart>
        <RhsChunk> End
<WhenPart>
    ::= When <NormalLhsBlock>
<RuleId>
    ::= Name
    |   String
```

Atributos de una Regla

Los atributos de una regla proveen una forma declarativa de influenciar en el comportamiento de la regla. Consiste en una lista de atributos disponibles que puede ser vacía.

```
<RuleAttributes>
    ::= <RuleAttributes> <RuleAttribute>
    |   ! Nothing
<RuleAttribute>
    ::= <Salience>
    |   <NoLoop>
    |   <AgendaGroup>
    |   <Duration>
    |   <ActivationGroup>
    |   <AutoFocus>
    |   <DateEffective>
    |   <Enabled>
    |   <LockOnActive>
<DateEffective>
    ::= 'date-effective' String
<Enabled>
    ::= 'enabled' Bool
    |   'enabled' <ParenChunk>
<Salience>
    ::= 'salience' Int
    |   'salience' <ParenChunk>
<NoLoop>
    ::= 'no-loop'
    |   'no-loop' Bool
<AutoFocus>
    ::= 'auto-focus'
    |   'auto-focus' Bool
<Duration>
    ::= 'duration' Int
    |   'duration' <ParenChunk>
<LockOnActive>
    ::= 'lock-on-active'
    |   'lock-on-active' Bool
<AgendaGroup>
    ::= 'agenda-group' String
<ActivationGroup>
    ::= 'activation-group' String
```

LHS

La parte izquierda de las reglas (Left Hand Side) es la sección donde se definen las condiciones.

```
<NormalLhsBlock>
    ::= <Lhs>
    |   <NormalLhsBlock> <Lhs>
    |   !Nothing
<Lhs>
    ::= <LhsOr>
<LhsOr>
    ::= '(' or <LhsAnd> ')'
    |   '(' or <LhsOr> <LhsAnd> ')'
    |   <LhsAnd>
    |   <LhsOr> or <LhsAnd>
<LhsAnd>
    ::= '(' and <LhsUnary> ')'
    |   '(' and <LhsAnd> <LhsUnary> ')'
    |   <LhsUnary>
    |   <LhsAnd> and <LhsUnary>
<LhsUnary>
    ::= <LhsExist>
    |   <LhsNot>
    |   <LhsEval>
    |   '(' <LhsOr> ')'
    |   <LhsPattern>
<LhsExist>
    ::= 'exists' <LhsOr>
<LhsNot>
    ::= 'not' <LhsOr>
<LhsEval>
    ::= 'eval' <ParenChunk>
<LhsPattern>
    ::= <FactBinding>
    |   <Fact>
<FactBinding>
    ::= <Label> <Fact>
    |   <Label> '(' <FactBindingExpression> ')'
<FactBindingExpression>
    ::= <Fact>
    |   <FactBindingExpression> or <Fact>
<Fact>
    ::= <PatternType> '(' ')'
    |   <PatternType> '(' <Constraints> ')'
<Constraints>
    ::= <Constraint>
    |   <Constraints> ',' <Constraint>
<Constraint>
    ::= <OrConstr>
<OrConstr>
    ::= <AndConstr>
    |   <OrConstr> '||' <AndConstr>
<AndConstr>
    ::= <UnaryConstr>
    |   <AndConstr> and <UnaryConstr>
<UnaryConstr>
```

```
    ::= 'eval' <ParenChunk>
    |   <FieldConstraint>
    |   '(' <OrConstr> ')'
<FieldConstraint>
    ::= <Label> <AccessorPath>
    |   <Label> <AccessorPath> <OrRestrConnective>
    |   <Label> <AccessorPath> Arrow <ParenChunk>
    |   <AccessorPath> <OrRestrConnective>
<Label>
    ::= Ampersand Name ':'
<OrRestrConnective>
    ::= <AndRestrConnective>
    |   <OrRestrConnective> '||' <AndRestrConnective>
<AndRestrConnective>
    ::= <ConstraintExpression>
    |   <AndRestrConnective> and <ConstraintExpression>
<ConstraintExpression>
    ::= <CompoundOperator>
    |   <SimpleOperator>
    |   '(' <OrRestrConnective> ')'
<SimpleOperator>
    ::= Equal <ExpressionValue>
    |   '>' <ExpressionValue>
    |   '>=' <ExpressionValue>
    |   '<' <ExpressionValue>
    |   '<=' <ExpressionValue>
    |   NotEqual <ExpressionValue>
    |   <OperatorKey> <ExpressionValue>
    |   <OperatorKey> <SquareChunk> <ExpressionValue>
    |   'not' <OperatorKey> <ExpressionValue>
    |   'not' <OperatorKey> <SquareChunk>
        <ExpressionValue>
<CompoundOperator>
    ::= 'in' '(' <ExpressionValue> ')'
    |   'in' '(' <CompoundOperator> ',' <ExpressionValue>
        ')'
    |   'not' 'in' '(' <ExpressionValue> ')'
    |   'not' 'in' '(' <CompoundOperator> ','
        <ExpressionValue> ')'
<OperatorKey>
    ::= Name
<ExpressionValue>
    ::= <AccessorPath>
    |   <LiteralConstraint>
    |   <ParenChunk>
<LiteralConstraint>
    ::= String
    |   Int
    |   Float
    |   Bool
    |   null
<PatternType>
```

```
    ::= Name
    |   <PatternType> '.' Name
    |   Name <DimensionDefinition>
    |   <PatternType> Name
<DimensionDefinition>
    ::= '[' ']'
<AccessorPath>
    ::= <AccessorElement>
    |   <AccessorPath> '.' <AccessorElement>
<AccessorElement>
    ::= Ampersand Name
    |   Name
    |   Ampersand Name <SquareChunk>
```

RHS

La sección derecha de una regla (Right Hand Side) es un bloque de código procedural a ser ejecutado; lo que se conoce como "acciones" de una regla.

```
<RhsChunk>
    ::= Then <Stm List>
<ParenChunk>
    ::= <ParenChunkData>
<ParenChunkData>
    ::= '(' ')'
    |   '(' <ParenChunkData> ')'
    |   '(' Int Equal Int ')'
<SquareChunk>
    ::= <SquareChunkData>
<SquareChunkData>
    ::= '[' ']'
    |   '[' <SquareChunkData> ']'
<Stm List>
    ::= <Statement>
    |   <Stm List> <Statement>
<Statement>
    ::= <Control Stm>
    |   <Normal Stm>
    |   <Local Var Decl>
<Control Stm>
    ::= if <Expression> <Stm List> endif
    |   if <Expression> <Then Stm> else <Stm List> endif
<Then Stm>
    ::= if <Expression> <Then Stm> else <Then Stm> endif
    |   <Stm List>
<Normal Stm>
    ::= <Statement Exp>
    |   <RuleEngineOperations>
<RuleEngineOperations>
    ::= Update '(' Ampersand Name ')'
```

```
    |   Modify '(' Ampersand Name ')'
    |   Retract '(' Ampersand Name ')'
    |   Insert '(' Ampersand Name ')'
    |   InsertLogical '(' Ampersand Name ')'
<Statement Exp>
    ::= <Qualified ID> '(' <Arg List Opt> ')'
    |   <Qualified ID> '(' <Arg List Opt> ')'
        <Methods Opt> <Assign Tail>
    |   <Qualified ID> <Assign Tail>
<Assign Tail>
    ::= Assign <Expression>
    |   '+=' <Expression>
    |   '-=' <Expression>
    |   '*=' <Expression>
    |   '/=' <Expression>
<Methods Opt>
    ::= <Methods Opt> <Method>
    |   !Null
<Method>
    ::= '.' Name
    |   '.' Name '(' <Arg List Opt> ')'
<Valid ID>
    ::= Ampersand Name
    |   Name
    |   FunctionCommandName
    |   RuleEngineReference
<Qualified ID>
    ::= <Valid ID> <Member List>
    |   NewOperator <Method>
<Member List>
    ::= <Member List> '.' Name
    |   !Zero or more
<Arg List Opt>
    ::= <Arg List>
    |   !Nothing
<Arg List>
    ::= <Arg List> ',' <Argument>
    |   <Argument>
<Argument>
    ::= <Expression>
<Expression>
    ::= <Conditional Exp> Assign <Expression>
    |   <Conditional Exp> '+=' <Expression>
    |   <Conditional Exp> '-=' <Expression>
    |   <Conditional Exp> '*=' <Expression>
    |   <Conditional Exp> '/=' <Expression>
    |   <Conditional Exp>
<Conditional Exp>
    ::= <Or Exp>
<Or Exp>
    ::= <Or Exp> or <And Exp>
    |   <And Exp>
```

```
<And Exp>
    ::= <And Exp> and <Logical Or Exp>
    |   <Logical Or Exp>
<Logical Or Exp>
    ::= <Logical Or Exp> '|' <Logical Xor Exp>
    |   <Logical Xor Exp>
<Logical Xor Exp>
    ::= <Logical Xor Exp> '^' <Logical And Exp>
    |   <Logical And Exp>
<Logical And Exp>
    ::= <Logical And Exp> and <Equality Exp>
    |   <Equality Exp>
<Equality Exp>
    ::= <Equality Exp> Equal <Compare Exp>
    |   <Equality Exp> NotEqual <Compare Exp>
    |   <Compare Exp>
<Compare Exp>
    ::= <Compare Exp> '<' <Shift Exp>
    |   <Compare Exp> '>' <Shift Exp>
    |   <Compare Exp> '<=' <Shift Exp>
    |   <Compare Exp> '>=' <Shift Exp>
    |   <Shift Exp>
<Shift Exp>
    ::= <Add Exp>
<Add Exp>
    ::= <Add Exp> '+' <Mult Exp>
    |   <Add Exp> '-' <Mult Exp>
    |   <Mult Exp>
<Mult Exp>
    ::= <Mult Exp> '*' <Unary Exp>
    |   <Mult Exp> '/' <Unary Exp>
    |   <Mult Exp> '%' <Unary Exp>
    |   <Unary Exp>
<Unary Exp>
    ::= '!' <Unary Exp>
    |   '-' <Unary Exp>
    |   <Object Exp>
<Object Exp>
    ::= <Method Exp>
<Method Exp>
    ::= <Method Exp> <Method>
    |   <Primary Exp>
<Primary Exp>
    ::= NewOperator <Non Array Type> '('<Arg List Opt>')'
    |   <Primary>
    |   '(' <Expression> ')'
<Primary>
    ::= <Valid ID>
    |   <Valid ID> '(' <Arg List Opt> ')'
    |   <LiteralConstraint>
<Local Var Decl>
    ::= <Qualified ID> <Variable Decs>
```

```
<Variable Decs>
    ::= <Variable Declarator>
    |   <Variable Decs> ',' <Variable Declarator>
<Variable Declarator>
    ::= Ampersand Name
    |   Ampersand Name Assign <Variable Initializer>
<Variable Initializer>
    ::= <Expression>
<Non Array Type>
    ::= <Qualified ID>
```

F.2. GeneXus Rule DSL

Para la construcción del lenguaje *GXRDSL - GeneXus Rule DSL* se toma como referencia el enfoque DSL [169] y el estándar SBVR [93].

Mediante el uso del enfoque DSL se busca lograr un lenguaje de definición de reglas de expresividad controlada que extienda al lenguaje "técnico" de reglas (detallado en la sección F.1).

Tomando ideas del estándar SBVR se busca lograr una especificación de reglas de negocio cercana al lenguaje natural para facilitar el entendimiento por parte del experto del negocio. En este sentido, uno de los mecanismos existentes para lograr una regla cercana al experto es desarrollar un glosario de términos del negocio o vocabulario para referenciar de forma no ambigua los diferentes elementos que conforman el dominio de la aplicación; comúnmente denominado modelo de objetos del negocio (BOM - Business Object Model).

En cierto sentido es un mapping entre una representación acorde al experto del negocio y el modelo de objetos asociado con la aplicación; el modelo de objetos ejecutable (XOM - Executable Object Model) acorde al personal técnico encargado del desarrollado de la aplicación.

En el contexto GeneXus la principal fuente de hechos y términos sobre los cuales se construyen reglas de negocio son los objetos transacción de tipo *BC - Business Component*, objetos estructurados (SDT - Structured Data Types), los atributos que los componen y sus relaciones.

Objeto RuleDSL

El objeto *RuleDSL* es justamente una forma de verbalizar las propiedades más importantes de los objetos de negocio para poder definir posteriormente reglas de negocio sobre dichos componentes. Se compone de las siguientes secciones:

- *Language Expression*: lenguaje para el experto del negocio.

- *Mapping Expression*: lenguaje técnico de reglas.
- *Scope*: alcance de la expresión.

La sección "*Scope*" indica el contexto en el cual la expresión es válida:

- *CONDITION*: la expresión es válida en la sección de condiciones.
- *CONSEQUENCE*: la expresión es válida en la sección de acciones.
- *KEYWORD*: la expresión es una palabra clave.
- *ANY*: la expresión es válida en cualquier sección.

El implantador define el lenguaje de modelado que contendrá los conceptos y reglas especificando internamente la transformación (mapping) al lenguaje técnico de reglas. El experto del negocio dispone de dichos bloques constructivos para definir reglas y políticas de negocio.

Funcionamiento

El objeto GeneXus *RuleDSL* en las columnas *Language Expression* y *Mapping Expression* define un conjunto de reglas de sustitución representadas de la forma:

$$patrón \rightarrow reemplazo$$

Para transformar una regla utilizando el esquema DSL al lenguaje técnico se realizan las siguientes operaciones (se utiliza el algoritmo de Markov [52]):

1. Se chequean las reglas de traducción del objeto *RuleDSL* en el orden que fueron definidas de arriba hacia abajo para verificar si alguno de los patrones se encuentran en la regla de entrada.
2. En caso negativo, el algoritmo finaliza.
3. Si se encuentra uno o más patrones, se reemplaza el texto izquierdo que "matchea" con su reemplazo para la primera regla que aplica.
4. Si la regla aplicada es la última, se termina el algoritmo.
5. Se vuelve al paso 1 del algoritmo.

El proceso de traducción lee la definición del DSL correspondiente, línea a línea y trata de "matchear" la secciones "*Language Expression*" – según el alcance – al lenguaje técnico que se detalla en la columna "*Mapping Expression*". Una vez que se obtiene la representación en formato técnico GeneXus, se utiliza el mecanismo de parsing (detallado en la sección F.1) para la validación y posterior traducción a la plataforma de ejecución.

Objetos RuleDSL

Se definieron diferentes lenguajes de alto nivel de abstracción (objetos *RuleDSL*) para utilizarse en los casos de uso que propone el Anexo G; se detallan a continuación.

Cliente

Objeto *dslCustomer000* de tipo *RuleDSL* utilizado en el caso de segmentación de clientes, sección G.2.

Language Expression	Mapping Expression	Scope
Customer gender is "{gender}" and age is less than '{age}'	&c : Customer(CustomerGender == "{gender}" , CustomerAge <= {age})	CONDITION
Customer age is greater than '{age}'	&c : Customer(CustomerAge > {age})	CONDITION
Customer gender between '{startAge}' and '{finishAge}'	&c : Customer(CustomerAge >= {startAge} , CustomerAge <= {finishAge})	CONDITION
Customer gender between '{startAge}' and '{finishAge}'	&c : Customer(CustomerAge >= {startAge} , CustomerAge <= {finishAge})	CONDITION
Customer Type is '{type}'	&c : Customer(CustomerTypeid == {type})	CONDITION
– No Identifier	&cId:CustomerId < 1	CONDITION
– From Montevideo	CountryId == 1,CityId == 1	CONDITION
There is a Customer	&c : Customer( )	CONDITION
Log : "{message}"	Msg("{message}")	CONSEQUENCE
Set Customer Type to '{value}'	&c.CustomerTypeid = {value}	CONSEQUENCE
Set Customer Maximum Allowed to '{value}'	&c.CustomerMaxAllowed = {value}	CONSEQUENCE
Update the fact : '{variable}'	update(&{variable})	CONSEQUENCE
do not loop	no-loop true	KEYWORD
priority {v}	salience {v}	KEYWORD

Producto

Objeto *dslProduct000* de tipo *RuleDSL* utilizado en el caso de uso de categorización de productos, sección G.2.

Language Expression	Mapping Expression	Scope
El Producto	&p : Product( )	CONDITION
- Diferente A Uruguay	Countryid < > 1	CONDITION
- De EEUU	Countryid == 4	CONDITION
- Juguete	Productcategoryid == 2	CONDITION
- De China	Countryid == 5	CONDITION
- De Uruguay	Countryid == 1	CONDITION
Mostrar Mensaje "{m}"	Msg("{m}")	CONSEQUENCE

Incremetar Producto Base en {b}	&p.Productprice = &p.Productprice + &p.Productbase * {b}	CONSEQUENCE
Asignar Precio Base	&p.Productprice = &p.Productbase	CONSEQUENCE
fin	end	KEYWORD
no reevaluar	no-loop true	KEYWORD
regla	rule	KEYWORD
hacer	then	KEYWORD
cuando	when	KEYWORD
habilitar	enabled true	KEYWORD
deshabilitar	enabled false	KEYWORD
vigencia "{dd-MMM-yyyy}"	date-effective "{dd-MMM-yyyy}"	KEYWORD
prioridad {p}	salience {p}	KEYWORD

Factura

Objeto *dslInvoice000* de tipo *RuleDSL* utilizado en el caso de uso de facturación, sección G.2.

Language Expression	**Mapping Expression**	**Scope**
- Cantidad de Líneas de Facura {operador} {valor}	InvoiceLine.Count {operador} {valor}	CONDITION
- Edad {operador} {valor}	Customerage {operador} {valor}	CONDITION
El Producto	&p : Product( )	CONDITION
- Tipo {t}	Customertypeid == {t}	CONDITION
- Juguete	Productcategoryid == 2	CONDITION
- De China	Countryid == 5	CONDITION
- De Uruguay	Countryid == 1	CONDITION
- del Cliente	Customerid == &c.Customerid	CONDITION
Cliente	&c : Customer( )	CONDITION
- Subtotal {operador} {valor}	Invoicesubtotal {operador} {valor}	CONDITION
Línea de Factura	&il: Invoice.Line( )	CONDITION
Factura	&i : Invoice( )	CONDITION
- Cantidad {operador} {valor}	&qty : Invoicelineqty {operador} {valor}	CONDITION
- Martes	&myDate : Invoicedate.Day == 2	CONDITION
- mismo Producto	Productid == &p.Productid	CONDITION
Producto	&p : Product( )	CONDITION
- De Montevideo	Countryid == 1 , Cityid == 1	CONDITION
- De Cliente	Customerid == &c.Customerid	CONDITION
- Fecha entre "{a}" y "{b}"	&date : Invoicedate >= "{a}" , Invoicedate <= "{b}"	CONDITION
- ProductId id	Productid == id	CONDITION
y	and	CONDITION
- Lunes	&myDate : Invoicedate.Day == 1	CONDITION
- Miercoles	&myDate : Invoicedate.Day == 3	CONDITION

- Fecha mayor a "{v}"	&date : Invoicedate > "{a}"	CONDITION
- Product Identifier desde {a} hasta {b}	Productid >= {a} , Productid <= {b}	CONDITION
Mostrar Mensaje "{m}"	Msg("{m}")	CONSEQUENCE
Incremetar Dto en {b}	&i.Invoicediscount = &i.Invoicesubtotal * {b}	CONSEQUENCE
Asignar Precio Base	&p.Productprice = &p.Productbase	CONSEQUENCE
Analizar Grupo de reglas "{g}"	RuleEngine.Focus("{g}")	CONSEQUENCE
Agregar un Error "{m}"	RuleEngine.AddError(1,"{m}")	CONSEQUENCE
Suspender Ejecución	RuleEngine.Halt()	CONSEQUENCE
Asignar Descuento {q}	&i.Invoicediscount = &i.Invoicesubtotal*{q}	CONSEQUENCE
Nueva línea de Factura	&il1 = new Invoice.Line()	CONSEQUENCE
Asignar Línea Id {id}	&il1.Invoicelineid = {id}	CONSEQUENCE
Asignar Producto Id {id}	&il1.Productid = {id}	CONSEQUENCE
Asignar Cantidad {qty}	&il1.Invoicelineqty = {qty}	CONSEQUENCE
Asignar Base {qty}	&il1.Invoicelineproductprice = {qty}	CONSEQUENCE
Asignar Taza {qty}	&il1.Invoicelinetaxes = {qty}	CONSEQUENCE
Asignar Dto {qty}	&il1.Invoicelinediscount = {qty}	CONSEQUENCE
Asignar Total {qty}	&il1.Invoicelineamount={qty}	CONSEQUENCE
Actualizar Contexto	insert(&il1)	CONSEQUENCE
Agregar Línea a Factura	&i.Line.add(&il1 , 0)	CONSEQUENCE
Acumular Descuento {q}	&i.Invoicediscount = &i.Invoicesubtotal * {q}	CONSEQUENCE
Uno Va Gratis	&il.InvoiceLineAmount = &il.InvoiceLineAmount - &il.InvoiceLineProductPrice	CONSEQUENCE
Inicializar Taza en {qty}	&il.Invoicelinetaxes = {qty}	CONSEQUENCE
fin	end	KEYWORD
no reevaluar	no-loop true	KEYWORD
regla	rule	KEYWORD
si	when	KEYWORD
entonces	then	KEYWORD
prioridad {p}	salience {p}	KEYWORD
Agrupar en "{g}"	agenda-group "{g}"	KEYWORD
habilitar	enabled true	KEYWORD
deshabilitar	enabled false	KEYWORD
Existe	exists	KEYWORD
No	not	KEYWORD
finalizar	end	KEYWORD

Procesos de Negocio

En el caso de uso relacionado con Workflow (sección G.2) se utilizan dos objetos de tipo *RuleDSL*.

Objeto *dslFlowEnglish*:

Language Expression	Mapping Expression	Scope
Request Description is Empty	&myRequest:Request(ReqDsc=="")	CONDITION
A Request	&myRequest:Request()	CONDITION
- Total greater than {v}	&tot:ReqTotAmt > {v}	CONDITION
Message "{m}"	Msg("{m}")	CONSEQUENCE
Set Authorization Status "{v}"	&myRequest.ReqAut = {v}	CONSEQUENCE
Stop Execution	RuleEngine.Halt()	CONSEQUENCE
Do not loop	no-loop true	KEYWORD
Low Priority	salience -10	KEYWORD
High Priority	salience 10	KEYWORD

Objeto *dslFlowJapanese*:

Language Expression	Mapping Expression	Scope
要請説明がむなしいです	&myRequest:Request(ReqDsc=="")	CONDITION
要請	&myRequest:Request()	CONDITION
-より素晴らしいことを合計してください {v}	&tot:ReqTotAmt > {v}	CONDITION
メッセージ "{m}"	Msg("{m}")	CONSEQUENCE
セットされた委任ステータス "{v}"	&myRequest.ReqAut = {v}	CONSEQUENCE
停止実行	RuleEngine.Halt()	CONSEQUENCE
輪にならないでください	no-loop true	KEYWORD
低いプライオリティー	salience -10	KEYWORD
最高水準プライオリティー	salience 10	KEYWORD

G. Escenarios de Uso

En esta sección se resumen los escenarios analizados utilizando el prototipo construido.

Inicialmente se detalla el negocio sobre el cual se construye el ejemplo. Posteriormente se explicitan los casos en donde se desea tener la posibilidad de agregar\modificar comportamiento en tiempo de ejecución y como aplica una solución utilizando reglas de negocio.

Para cada ejemplo se especifica el juego de reglas que se desea evaluar. Se crean las reglas en el lenguaje de especificación de reglas de negocio en formato técnico y donde aplique, se detalla el formato "alto nivel" de abstracción orientado al experto del negocio. Se agrega el código necesario para poder evaluar los juegos de reglas y por último se resume la evaluación de cada escenario detallando cómo se resuelve el problema.

G.1. Descripción del negocio

El ejemplo canónico se basa en una pequeña aplicación de compras de una empresa. Las entidades principales son Cliente (Customer), Tipo De Cliente (CustomerType), Producto (Product), Categoría De Producto (ProductCategory) y Factura (Invoice) como muestra la Figura G.1:

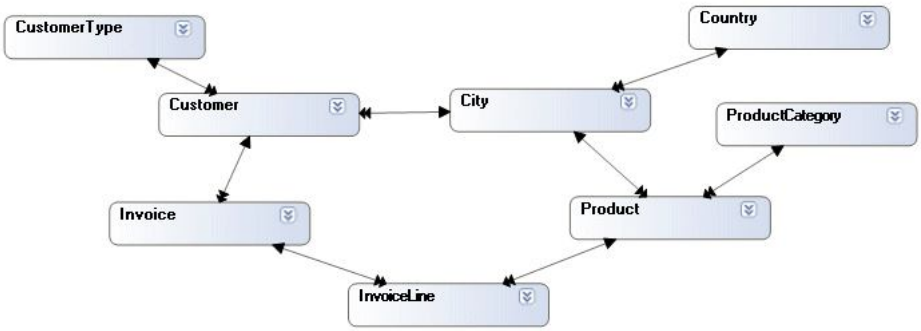


Figura G.1.: Ejemplo canónico

Las operaciones fundamentales que ocurren son:

- Registro de Clientes.

- Ingreso y categorización de Productos.
- Registro de Facturas.

Cada entidad definida anteriormente es representada en el ecosistema GeneXus por un objeto de tipo *Business Component*.

Se parte de una aplicación GeneXus que implementa la persistencia de los diferentes objetos del negocio sin indicar ningún tipo de comportamiento extra y se define una interfaz de usuario básica para poder interactuar con las entidades.

Se pretende agregar comportamiento según sea necesario, sin modificar el código de la aplicación utilizando reglas de negocio. Una breve descripción sobre los casos de uso se detalla a continuación:

- *Cliente*: segmentación y asignación del máximo de compra permitido por cliente.
- *Producto*: inicialización del precio de un producto utilizando diferentes consideraciones como por ejemplo, categoría y procedencia.
- *Factura*: aplicar diferentes tipos de validaciones, descuentos y promociones cuando se crean facturas.
- *Workflow*: separar ciertas decisiones de un flujo de negocio utilizando reglas.
- *OAV* y *modelos universales de datos*: a partir de modelos genéricos que se utilizan para representar cualquier tipo de Entidad, agregar comportamiento y validaciones sobre las entidades, sus propiedades y relaciones.
- *Características Avanzadas*: experimentación con características avanzadas de los motores de evaluación como por ejemplo inferencia y *Truth Maintenance* (inserción lógica) [142, 207].

G.2. Escenarios

Los escenarios implementados durante la construcción del prototipo se detallan a continuación:

- *Escenario 0 – Hola Mundo*: caso base para comprender el funcionamiento del motor de evaluación de reglas en el contexto GeneXus.
- *Escenario 1 – Cliente*: se definen reglas de negocio sobre el ingreso y actualización de clientes.
- *Escenario 2 – Producto*: al igual que en el ejemplo anterior se utiliza el mismo ejemplo para definir reglas de negocio sobre el ingreso y actualización de productos.

- *Escenario 3 – Factura*: se definen reglas sobre el ingreso de facturas, involucrando y relacionando varios objetos en su definición como ser *Cliente*, *Producto*, *Factura* y *líneas de Factura*.
- *Escenario 4 – Workflow*: se explora el uso de reglas de negocio integrado en procesos de negocio.
- *Escenario 5 – OAV*: se experimenta la inclusión de reglas de negocio en modelos de datos genéricos.
- *Escenario 6 – Interacción con código GeneXus*: se investiga el uso de objetos GeneXus desde la evaluación dinámica de reglas.
- *Escenario 7 – Modelo Universal de Datos*: se extiende el caso de uso *OAV* definiendo reglas de negocio sobre modelos de datos universales.
- *Escenario 8 – Uso de Operaciones*: se investiga el uso de objetos GeneXus desde la evaluación dinámica de reglas.
- *Escenario 9 – "Truth Maintenance"*: se valida el uso del motor de evaluación de reglas como mecanismo de inferencia o razonamiento.

Escenario 0 – Hola Mundo

El escenario *Hola Mundo* consiste en la prueba inicial a realizar para evaluar el funcionamiento del motor de reglas, entender cómo se debe instanciar el motor de evaluación y asignar el juego de reglas y objetos sobre los cuales se quiere aplicar las reglas.

En términos generales el funcionamiento es el siguiente:

- Definición del juego de reglas utilizando objetos de tipo *Ruleset* y *RuleDSL* en el IDE GeneXus o en el editor externo de reglas.
- Desarrollar código procedural para interactuar con el motor de evaluación; que incluye:
  - Instanciación del motor de evaluación de reglas.
  - Asignación de un juego de reglas.
  - Inserción de los objetos de dominio en la memoria de trabajo.
  - Ejecución del método que evalúa las reglas (método *FireAllRules* del objeto *RulesetEngine*).

El ejemplo *Hola Mundo* (detallado en la figura G.2) consta de un objeto simple denominado `MyMessage` con dos propiedades `Status` de tipo *Boolean* y `Message` de tipo *String*.

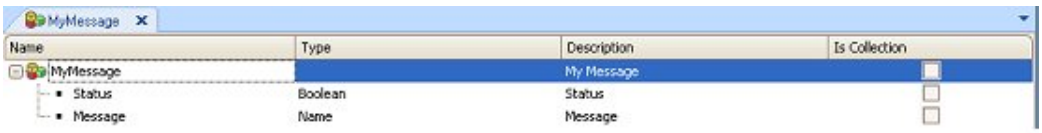


Figura G.2.: Objeto Estructurado - MyMessage

y las siguientes reglas utilizando el lenguaje de definición de reglas técnico:

```
rule "Hello World"
  when
    &m : MyMessage(Status == true, &theMessage:Message)
  then
    Msg( &theMessage );
    &m.Message = "Goodbye cruel world"
    &m.Status = false
    update( &m )
end

rule "GoodBye"
  when
    MyMessage(Status == false, &theMessage:message)
  then
    Msg( &theMessage )
end
```

La parte izquierda de la regla `Hello World` detalla que la misma será activada por cada objeto de tipo `MyMessage` que se encuentre en la memoria de trabajo cuyo `Status` sea verdadero (*true*); proyectando el objeto en la variable `&m` y la propiedad `Message` en la variable `&theMessage`.

La sección de consecuencias de la misma regla muestra por pantalla que la regla se ejecutó referenciando el valor de la variable `&theMessage`, se modifican las propiedades `Mensaje` y `Status` del objeto `MyMessage` utilizando la referencia `&m` y posteriormente actualiza el objeto en la memoria de trabajo utilizando la operación `update`.

La regla `Goodbye` es muy similar a la anterior pero en este caso se seleccionan aquellos objetos `MyMessage` en donde el `Status` se corresponde con el valor *false*, proyectando el valor de la propiedad `Message` en la variable `&theMessage` que luego se muestra por pantalla (utilizando la función `Msg`) en la sección de consecuencias de la regla.

Supongamos que se crea un objeto `MyMessage` con las siguientes propiedades:

```
&message = new()
&message.Message = "Hello World"
&message.Status = true
```

El código procedural para interactuar con el motor de evaluación de reglas es el siguiente:

```
&myRulesetEngine.Ruleset(&Ruleset)
&myRulesetEngine.Insert(&message)
&myRulesetEngine.FireRules()
```

siendo `&myRulesetEngine` una instancia del motor de evaluación, `&Ruleset` una variable que contiene el nombre del juego de reglas a utilizar.

El resultado esperado de la evaluación de reglas es el siguiente:

- Activación de la regla "`Hello World`" después de determinar que las condiciones de la regla aplican sobre al menos un objeto en la memoria de trabajo del motor de evaluación ("matcheo" de patrones exitoso).
- Ejecución de la parte derecha de la regla "`Hello World`".
  - Se detalla por salida estándar un mensaje (uso de la función `Msg`).
  - Se actualizan propiedades del propio objeto.
  - Se indica al motor que el objeto ha sido actualizado (función `update`).
- Se repite el ciclo de evaluación de reglas al ejecutarse la operación `update`.
- Activación de la regla "`GoodBye`".
- Ejecución de la parte derecha de la regla "`Goodbye`".
  - Se detalla por salida estándar un mensaje, nuevamente utilizando la función `Msg`.
- Se repite el ciclo de evaluación, no existiendo activaciones para los objetos en la memoria de trabajo.
- Fin de ejecución del motor de reglas; se devuelve el control procedural al objeto llamador.

Consideraciones

Se valida la ejecución hacia adelante del motor de inferencia (mecanismo de inferencia *forward chaining*), la habilidad del motor para evaluar, agendar, activar y disparar en secuencia las diferentes consecuencias de las reglas y también se considera los cambios que se realicen a los objetos durante ésta ejecución.

Escenario 1 – Cliente

El escenario relacionado con el mantenimiento de clientes tiene como objetivo poder realizar modificaciones en términos de restricciones y validaciones una vez que la aplicación ya se encuentra en ejecución. El cliente está representado por dos objetos de negocio denominados `Customer` y `CustomerType` como detalla la Figura G.3:

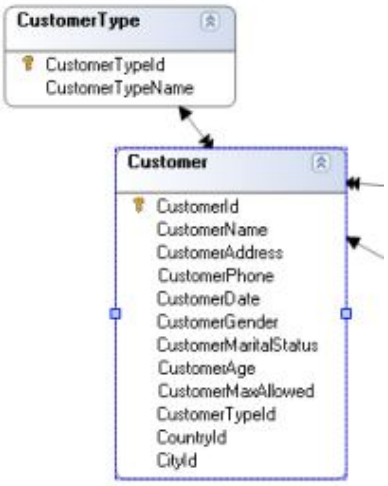


Figura G.3.: Objeto Cliente y Tipo de Cliente

Un cliente (`Customer`) está formado por las siguientes propiedades:

- *CustomerId*: Identificador.
- *CustomerName*: Nombre.
- *CustomerAddress*: Dirección.
- *CustomerPhone*: Teléfono.
- *CustomerDate*: Fecha de nacimiento.
- *CustomerGender*: Género.
- *CustomerMaritalStatus*: Estado matrimonial.
- *CustomerAge*: Edad.
- *CustomerMaxAllowed*: Máximo de compras permitido.

A su vez el mismo tiene asociado un tipo de cliente (`CustomerType`):

- *CustomerTypeId*: Identificador.

- *CustomerTypeName*: Descripción del tipo de cliente.

Los tipos de clientes pueden ser A, B o C referenciados por los identificadores 1, 2 y 3 respectivamente. Se le asigna *Ciudad* y *País* de Origen:

- *CityId*: Identificador de ciudad.
- *CountryId*: Identificador de país.

Inicialización

En tiempo de actualización del cliente se desea segmentarlo en los tipos A, B, C utilizando las siguientes reglas y también se desea poder modificar fácilmente dichas definiciones en el futuro.

- *Regla 1*: Todos los clientes por defecto son de tipo B.

```
rule "Todos los clientes son B"
  no-loop true
  when
    &c : Customer(CustomerTypeId=="")
  then
    Msg( "Todos los clientes son B" )
    &c.CustomerTypeId = 2 // B
end
```

- *Regla 2*: Clientes masculinos de hasta 25 años son de tipo A.

```
rule "Clientes masculinos de hasta 25 años"
  no-loop true
  when
    &c : Customer( Customergender == "M" ,
                   Customerage <= 25 )
  then
    Msg("Clientes masculinos de hasta 25 años son de tipo A.")
    &c.Customertypeid = 1 // A
end
```

- *Regla 3*: Clientes femeninos de hasta 30 años son de tipo A.

```
rule "Clientes femeninos de hasta 30 años"
  no-loop true
  when
    &c : Customer( Customergender == "F",
                   Customerage <= 30 )
  then
    Msg("Clientes femeninos de hasta 30 son de tipo A.")
    &c.Customertypeid = 1 // A
end
```

- *Regla 4*: Clientes de más de 70 años son de tipo C.

```
rule "Clientes de mas de 70 años"
  no-loop true
  when
    &c : Customer( Customerage > 70 )
  then
    Msg("Clientes de mas de 70 son de tipo C.")
    &c.Customertypeid = 3 // C
end
```

Modificaciones

Debido a cambios en la realidad se desea aplicar los siguientes cambios.

Disminuir la franja etárea que segmenta a los clientes masculinos de tipo A de 30 a 20 años. Se modifica la Regla 2 de la siguiente manera:

- *Regla 2'*: Clientes masculinos de hasta 20 años son de tipo A.

```
rule "Clientes masculinos de hasta 20 años"
  no-loop true
  when
    &c : Customer( Customergender == "M", Customerage <= 20 )
  then
    Msg("Clientes masculinos de hasta 20 son de tipo A.")
    &c.Customertypeid = 1 // A
end
```

Agregar una nueva segmentación, aquellos Clientes entre 50 y 60 años pasan a pertenecer al tipo C; se agrega una nueva regla:

- *Regla 5*: Clientes entre 50 y 60 años asignar el tipo C.

```
rule "Clientes entre 50 y 60 años"
  no-loop true
  when
    &c : Customer( Customerage >= 50 , Customerage <= 60 )
  then
    Msg("Clientes entre 50 y 60 asignar el tipo C.")
    &c.Customertypeid = 3 // C
    update(&c)
end
```

Crear nuevas restricciones sobre otros términos del dominio. Para aquellos clientes que sean de tipo C asegurarse que el máximo permitido de compra sea 300. Se agrega la siguiente regla:

- *Regla 6*: Clientes de tipo C asignar 300 como máximo de compras.

```
rule "Clientes C tiene un máximo definido de compras"
  enabled true
  no-loop true
  when
    &c : Customer( Customertypeid == 3 )
  then
    Msg("Clientes de tipo C asignar 300 como máximo de compras.")
    &c.Customermaxallowed = 300.00
end
```

Evaluación

Se crea un procedimiento de ejemplo para dar de alta dos clientes con los siguientes datos iniciales:

```
Sub 'variables01'  // cliente femenino de 30 años
  &CustomerName = "Customer " + &CustomerId.ToString().Trim()
  &CustomerAddress = "Address " + &CustomerId.ToString().Trim()
  &CustomerAge = 30
  &CustomerDate = YMDtoD( 1978,9,7 )
  &CustomerGender = Gender.Female
  &CustomerMaritalStatus = MaritalStatus.Single
  &CustomerPhone = "9009090"
  &CustomerMaxAllowed = 100
  &CustomerTypeId = 1
  &CountryId = 1
  &CityId = 1
EndSub

Sub 'variables02'
  // cliente masculino de 55 años
  &CustomerName = "Customer " + &CustomerId.ToString().Trim()
  &CustomerAddress = "Address " + &CustomerId.ToString().Trim()
  &CustomerAge = 55
  &CustomerDate = YMDtoD( 1953,9,7 )
  &CustomerGender = Gender.Male
  &CustomerMaritalStatus = MaritalStatus.Single
  &CustomerPhone = "9001010"
  &CustomerMaxAllowed = 99
  &CustomerTypeId = 1
  &CountryId = 1
  &CityId = 1
EndSub
```

Se ejecuta el escenario utilizando los 2 juegos de reglas definidos; inicialmente con las reglas que se detallan en la sección de inicialización obteniendo el siguiente resultado sobre el primer cliente:

```
Clientes femeninos de hasta 30 son de tipo A.
```

El estado del cliente se detalla a continuación asignándose el tipo de cliente *A*:

```
<Customer xmlns="brsamplexev101">
  <CustomerId>152</CustomerId>
  <CustomerName>Customer 151</CustomerName>
  <CustomerAddress>Address 151</CustomerAddress>
  <CustomerPhone>9009090</CustomerPhone>
  <CustomerDate>1978-09-07</CustomerDate>
  <CustomerGender>F</CustomerGender>
  <CustomerMaritalStatus>S</CustomerMaritalStatus>
  <CustomerAge>30</CustomerAge>
  <CustomerMaxAllowed>100.00</CustomerMaxAllowed>
  <CustomerTypeId>1</CustomerTypeId>
  <CustomerTypeName>A</CustomerTypeName>
  <CountryId>1</CountryId>
  <CityId>1</CityId>
</Customer>
```

Para el segundo cliente no se ejecutan reglas:

```
<Customer xmlns="brsamplexev101">
  <CustomerId>153</CustomerId>
  <CustomerName>Customer 152</CustomerName>
  <CustomerAddress>Address 152</CustomerAddress>
  <CustomerPhone>9001010</CustomerPhone>
  <CustomerDate>1953-09-07</CustomerDate>
  <CustomerGender>M</CustomerGender>
  <CustomerMaritalStatus>S</CustomerMaritalStatus>
  <CustomerAge>55</CustomerAge>
  <CustomerMaxAllowed>99.00</CustomerMaxAllowed>
  <CustomerTypeId>1</CustomerTypeId>
  <CustomerTypeName>A</CustomerTypeName>
  <CountryId>1</CountryId>
  <CityId>1</CityId>
</Customer>
```

Sin modificar el juego de datos, se hacen las modificaciones sobre las reglas que detalla la sección *Modificaciones* (sección G.2) y ejecuta nuevamente. Para el primer cliente se ejecutan exactamente las mismas reglas mientras que para el segundo cliente aplican las nuevas restricciones:

```
Clientes entre 50 y 60 asignar el tipo C.
Clientes de tipo C asignar 300 como máximo de compras.
```

El segundo cliente al finalizar la evaluación de reglas de negocio tiene el siguiente estado:

```
<Customer xmlns="brsamplexev101">
  <CustomerId>155</CustomerId>
  <CustomerName>Customer 154</CustomerName>
```

```
  <CustomerAddress>Address 154</CustomerAddress>
  <CustomerPhone>9001010</CustomerPhone>
  <CustomerDate>1953-09-07</CustomerDate>
  <CustomerGender>M</CustomerGender>
  <CustomerMaritalStatus>S</CustomerMaritalStatus>
  <CustomerAge>55</CustomerAge>
  <CustomerMaxAllowed>300.00</CustomerMaxAllowed>
  <CustomerTypeId>3</CustomerTypeId>
  <CustomerTypeName>C</CustomerTypeName>
  <CountryId>1</CountryId>
  <CityId>1</CityId>
</Customer>
```

Lenguaje de Alto nivel

Utilizando el segundo juego de reglas, se desarrolla un lenguaje de alto nivel de abstracción en formato inglés para experimentar con una definición de reglas de negocio acorde al experto del negocio. Para ello se define un objeto *RuleDSL* de nombre *dslCustomer000* con las siguientes expresiones válidas del lenguaje y su correspondiente traducción al lenguaje técnico de reglas:

Language Expression	**Mapping Expression**	**Scope**
Customer gender is "{gender}" and age is less than '{age}'	&c : Customer(CustomerGender == "{gender}" , CustomerAge <= {age})	CONDITION
Customer age is greater than '{age}'	&c : Customer(CustomerAge > {age})	CONDITION
Customer gender between '{startAge}' and '{finishAge}'	&c : Customer(CustomerAge >= {startAge} , CustomerAge <= {finishAge})	CONDITION
Customer gender between '{startAge}' and '{finishAge}'	&c : Customer(CustomerAge >= {startAge} , CustomerAge <= {finishAge})	CONDITION
Customer Type is '{type}'	&c : Customer(CustomerTypeid == {type})	CONDITION
– No Identifier	&cId:CustomerId < 1	CONDITION
– From Montevideo	CountryId == 1,CityId == 1	CONDITION
There is a Customer	&c : Customer( )	CONDITION
Log : "{message}"	Msg("{message}")	CONSEQUENCE
Set Customer Type to '{value}'	&c.CustomerTypeid = {value}	CONSEQUENCE
Set Customer Maximum Allowed to '{value}'	&c.CustomerMaxAllowed = {value}	CONSEQUENCE
Update the fact : '{variable}'	update(&{variable})	CONSEQUENCE
do not loop	no-loop true	KEYWORD
priority {v}	salience {v}	KEYWORD

El juego de reglas detallado anteriormente se puede reescribir de la siguiente manera:

```
// indico que se va a utilizar el lenguaje detallado anteriormente
expander dslCustomer000
```

```
rule "Todos los clientes de UY son B"
  do not loop
  priority 100
  when
    There is a Customer
    - From Montevideo
  then
    Log : "Todos los clientes de UY son B"
    Set Customer Type to '2'// B
end

rule "Clientes masculinos de hasta 20 años son de tipo A."
  do not loop
  when
    Customer gender is "M" and age is less than '20'
  then
    Log : "Clientes masculinos de edad menor a 20 son de tipo A."
    Set Customer Type to '1'// A
end

rule "Clientes femeninos de hasta 30 años son de tipo A."
  do not loop
  when
    Customer gender is "F" and age is less than '30'
  then
    Log : "Clientes femeninos de hasta 30 son de tipo A."
    Set Customer Type to '1'// A
  end

rule "Clientes de mas de 70 años son de tipo C."
  do not loop
  when
    Customer age is greater than '70'
  then
    Log : "Clientes de mas de 70 son de tipo C."
    Set Customer Type to '3'// C
end

rule "Clientes entre 50 y 60 años asignar el tipo C."
  do not loop
  when
    Customer gender between '50' and '60'
  then
    Log : "Clientes entre 50 y 60 asignar el tipo C."
    Set Customer Type to '3'// C
    Update the fact : 'c'
end

rule "Clientes de tipo C asignar 300 como máximo de compras."
  do not loop
  when
```

```
    Customer Type is '3'
  then
    Log : "Clientes de tipo C asignar 300 como máximo de compras."
    Set Customer Maximum Allowed to '300.00'
end
```

La ejecución es equivalente al caso detallado anteriormente (sección G.2) dado que la definición del *RuleDSL* simplemente permite especificar las reglas a un nivel de abstracción mayor. Internamente cuando se salva el juego de reglas, se genera la traducción al lenguaje técnico GeneXus ya detallado. También se genera la representación del lenguaje de reglas en la plataforma de ejecución.

Consideraciones

Las reglas definidas se pueden analizar en forma centralizada en la tabla de decisión de la Figura G.4.

	Cursos de Acción	Reglas					
		1	2	3	4	5	6
Condiciones	CustomerGender		Male	Female			
	CustomerAge		<=20	<=30	>70	between(50,60)	
	CustomerTypeId						C
Acciones	Asignar CustomerType B	X					
	Asignar CustomerType A		X	X			
	Asignar CustomerType C				X	X	
	Asignar máximo compras permitido						X

Figura G.4.: Tabla de decisión del Escenario - Cliente

Notar que los cambios realizados no solo impactan a las reglas definidas inicialmente sino que también se realizan nuevas segmentaciones y restricciones que aplican sobre otros términos definidos en el dominio sin haber modificado una línea de código del programa que ejecuta.

Escenario 2 – Producto

El escenario relacionado con los productos tiene como objetivo inicializar el precio de un producto en base a diferentes restricciones relacionadas con la categoría y procedencia de los productos. Es muy similar al escenario anterior relacionado con el "*Cliente*"; aunque en este caso se evalúan diferentes tipos de expresiones – en particular reglas de tipo *Producción* – para realizar los cálculos que detallan las reglas de negocio.

El Producto está representado por dos objetos de negocio denominados `Product` y `ProductCategory` como detalla la Figura G.5.

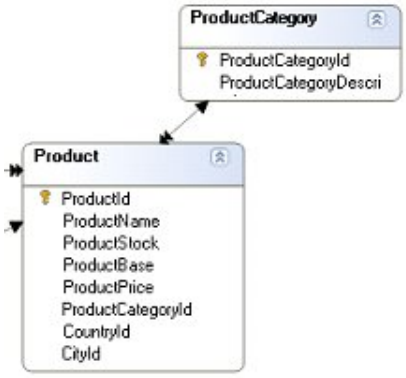


Figura G.5.: Objeto Producto y Categoría

Un Producto (`Product`) se compone de las siguientes propiedades:

- *ProductId*: Identificador.
- *ProductName*: Nombre.
- *ProductStock*: Stock disponible.
- *ProductBase*: Costo base del producto.
- *ProductPrice*: Costo del producto (aplica diferentes cálculos).

A su vez el mismo tiene asociada una categoría (`ProductCategory`):

- *CategoryId*: Identificador.
- *CategoryDescription*: Descripción de categoría.

Se le asigna Ciudad y País de Origen:

- *CityId*: Identificador de ciudad.
- *CountryId*: Identificador de país.

Inicialización

En tiempo de actualización del producto se aplican ciertas restricciones sobre el precio base para obtener el precio sugerido.

- *Regla 1*: Todos los productos que no son de Uruguay, aplicar un 1 % de recargo para promocionar el consumo interno.

```
rule "Productos <> Uruguay, aplicar 1% de recargo"
  no-loop true
  when
    &p : Product( Countryid <> 1 )
  then
    Msg("Productos que no son de Uruguay, aplicar un 1% de recargo")
    &p.Productprice = &p.Productprice + &p.Productbase * 0.01
end
```

- *Regla 2*: Todos los productos de Estados Unidos, aplicar un 5 % de recargo debido a los aranceles.

```
rule "Productos de USA, aplicar recargo 5%"
  no-loop true
  when
    &p : Product( Countryid == 4 )
  then
    Msg("Productos de Estados Unidos, aplicar un 5% de recargo")
    &p.Productprice = &p.Productprice + &p.Productbase * 0.05
end
```

Modificaciones

Debido a cambios en la realidad se desea aplicar los siguientes cambios. Todos los productos Chinos de tipo Juguete aplicar un arancel del 20 %; mientras que los juguetes uruguayos no aplican arancel.

Se agregan dos reglas para modelar las nuevas restricciones:

- *Regla 3*: Todos los productos que no son de Uruguay, aplicar un 1 % de recargo para promocionar el consumo interno.

```
rule "Juguetes Uruguayos, no aplicar arancel"
  no-loop true
  when
    &p : Product( Countryid == 1 , Productcategoryid == 2 )
  then
    Msg("Juguetes Uruguayos, no aplicar arancel")
    &p.Productprice = &p.Productbase
end
```

- *Regla 4*: Juguetes Chinos, aplicar un 20 % de arancel.

```
rule "Juguetes Chinos, aplicar arancel"
  no-loop true
  when
    &p : Product( Countryid == 5 , Productcategoryid == 2 )
  then
    Msg("Juguetes Chinos, aplicar un 20% de arancel")
    &p.Productprice = &p.Productprice + &p.Productbase * 0.2
end
```

Evaluación

Se crea un procedimiento que da de alta tres productos con los siguientes datos iniciales:

```
Sub 'variables01'
  // Producto de tipo juguete de Uruguay
  &ProductName = "Product " + &ProductId.ToString().Trim()
  &ProductPrice = 100
  &ProductStock = 100
  &ProductBase = 100
  &ProductCategoryId = 2 // Toys
  &CountryId = 1 // Uruguay
  &CityId = 2
EndSub

Sub 'variables02'
  // Producto de tipo juguete de USA
  &ProductName = "Product " + &ProductId.ToString().Trim()
  &ProductPrice = 200
  &ProductStock = 200
  &ProductBase = 200
  &ProductCategoryId = 2 // Toys
  &CountryId = 4 // USA
  &CityId = 1
EndSub

Sub 'variables03'
  // Producto de tipo juguete de China
  &ProductName = "Product " + &ProductId.ToString().Trim()
  &ProductPrice = 50
  &ProductStock = 50
  &ProductBase = 50
  &ProductCategoryId = 2 // Toys
  &CountryId = 5 // China
  &CityId = 1
EndSub
```

Se ejecuta la aplicación de prueba con el juego de reglas inicial detallado en la sección *Inicialización* obteniendo el siguiente resultado sobre el primer producto: no se ejecutan regla, permaneciendo incambiado el objeto cuando finaliza la evaluación:

```
<Product xmlns="brsamplexev101">
  <ProductId>240</ProductId>
  <ProductName>Product 239</ProductName>
  <ProductStock>100.00</ProductStock>
  <ProductBase>100.00</ProductBase>
  <ProductPrice>100.00</ProductPrice>
  <ProductCategoryId>2</ProductCategoryId>
  <ProductCategoryDescription>Toys</ProductCategoryDescription>
  <CountryId>1</CountryId>
  <CityId>2</CityId>
```

```
</Product>
```

Para el caso del segundo producto se ejecutan las siguientes reglas:

```
Productos de Estados Unidos, aplicar un 5% de recargo
Productos que no son de Uruguay, aplicar un 1% de recargo
```

El estado del producto se detalla a continuación una vez ejecutadas las consecuencias de las reglas:

```
<Product xmlns="brsamplexev101">
  <ProductId>241</ProductId>
  <ProductName>Product 240</ProductName>
  <ProductStock>200.00</ProductStock>
  <ProductBase>200.00</ProductBase>
  <ProductPrice>212.00</ProductPrice>
  <ProductCategoryId>2</ProductCategoryId>
  <ProductCategoryDescription>Toys</ProductCategoryDescription>
  <CountryId>4</CountryId>
  <CityId>1</CityId>
</Product>
```

Para el tercer producto se ejecuta la regla:

```
Productos que no son de Uruguay, aplicar un 1% de recargo
```

por lo que el estado del producto al finalizar la evaluación es el siguiente:

```
<Product xmlns="brsamplexev101">
  <ProductId>242</ProductId>
  <ProductName>Product 241</ProductName>
  <ProductStock>50.00</ProductStock>
  <ProductBase>50.00</ProductBase>
  <ProductPrice>50.50</ProductPrice>
  <ProductCategoryId>2</ProductCategoryId>
  <ProductCategoryDescription>Toys</ProductCategoryDescription>
  <CountryId>5</CountryId>
  <CityId>1</CityId>
</Product>
```

Sobre el mismo juego de datos se agregan las reglas que se detalla en la sección *Modificaciones* y ejecuta nuevamente. Para el primer cliente comienza a ser válida la regla:

```
Juguetes Uruguayos, no aplicar arancel
```

obteniéndose el siguiente resultado:

```
<Product xmlns="brsamplexev101">
  <ProductId>243</ProductId>
  <ProductName>Product 242</ProductName>
  <ProductStock>100.00</ProductStock>
  <ProductBase>100.00</ProductBase>
  <ProductPrice>100.00</ProductPrice>
  <ProductCategoryId>2</ProductCategoryId>
  <ProductCategoryDescription>Toys</ProductCategoryDescription>
  <CountryId>1</CountryId>
  <CityId>2</CityId>
</Product>
```

Para el segundo producto se ejecutan exactamente las mismas reglas mientras que para el tercer producto se ejecuta una nueva regla de negocio, implicando la actualización del precio base a un nuevo valor.

```
Juguetes Chinos, aplicar un 20% de arancel
Productos que no son de Uruguay, aplicar un 1% de recargo
```

El tercer producto al finalizar la evaluación de reglas define los siguientes valores:

```
<Product xmlns="brsamplexev101">
  <ProductId>245</ProductId>
  <ProductName>Product 244</ProductName>
  <ProductStock>50.00</ProductStock>
  <ProductBase>50.00</ProductBase>
  <ProductPrice>60.50</ProductPrice>
  <ProductCategoryId>2</ProductCategoryId>
  <ProductCategoryDescription>Toys</ProductCategoryDescription>
  <CountryId>5</CountryId>
  <CityId>1</CityId>
</Product>
```

Lenguaje de Alto nivel

Utilizando el segundo juego de reglas se desarrolla un lenguaje de alto nivel de abstracción que utiliza construcciones similares al "español". Notar que en este caso se utilizan varias palabras clave para traducir desde el formato estándar de definición de regla y lograr una representación de reglas similar al español. Es decir, a partir de la definición estándar de una regla de negocio que consta de las siguientes secciones:

```
rule
  when
    <<conditions>>
  then
    <<actions>>
end
```

se define un objeto *RuleDSL* de nombre *dslProduct000*, donde se centraliza la definición de los principales bloques constructivos necesarios para lograr definir una regla de negocio con un formato similar al lenguaje "español" como es el siguiente ejemplo:

```
regla
  cuando
    <<condiciones>>
  entonces
    <<acciones>>
fin
```

Se utilizan las siguientes expresiones válidas del lenguaje; notar que en este caso se agregan varias palabras clave (alcance *keyword* en la tabla a continuación) para lograr un lenguaje de reglas específico para hispanohablantes.

Language Expression	**Mapping Expression**	**Scope**
El Producto	&p : Product( )	CONDITION
- Diferente A Uruguay	Countryid < > 1	CONDITION
- De EEUU	Countryid == 4	CONDITION
- Juguete	Productcategoryid == 2	CONDITION
- De China	Countryid == 5	CONDITION
- De Uruguay	Countryid == 1	CONDITION
Mostrar Mensaje "{m}"	Msg("{m}")	CONSEQUENCE
Incremetar Producto Base en {b}	&p.Productprice = &p.Productprice + &p.Productbase * {b}	CONSEQUENCE
Asignar Precio Base	&p.Productprice = &p.Productbase	CONSEQUENCE
fin	end	KEYWORD
no reevaluar	no-loop true	KEYWORD
regla	rule	KEYWORD
hacer	then	KEYWORD
cuando	when	KEYWORD
habilitar	enabled true	KEYWORD
deshabilitar	enabled false	KEYWORD
vigencia "{dd-MMM-yyyy}"	date-effective "{dd-MMM-yyyy}"	KEYWORD
prioridad {p}	salience {p}	KEYWORD

Dicho lenguaje permite definir las reglas de la siguiente forma:

```
// indico que utilizo un lenguaje de alto nivel
expander dslProduct000

regla "Productos que no son de Uruguay, 1% de recargo"
  habilitar
  no reevaluar
  cuando
    El Producto
```

```
    - Diferente A Uruguay
  hacer
    Mostrar Mensaje "Productos que no son de Uruguay, 1% de recargo"
    Incremetar Producto Base en 0.01
fin

regla "Productos de Estados Unidos, 5% de recargo"
  no reevaluar
  cuando
    El Producto
    - De EEUU
  hacer
    Mostrar Mensaje "Productos de Estados Unidos, 5% de recargo"
    Incremetar Producto Base en 0.05
fin

regla "Juguetes Uruguayos, no aplicar arancel"
  no reevaluar
  cuando
    El Producto
    - Juguete
    - De Uruguay
  hacer
    Mostrar Mensaje "Juguetes Uruguayos, no aplicar arancel"
    Asignar Precio Base
fin

regla "Juguetes Chinos, 20% de arancel"
  prioridad 10
  no reevaluar
  vigencia "08-sep-2009"
  cuando
    El Producto
    - Juguete
      - De China
  hacer
    Mostrar Mensaje "Juguetes Chinos, 20% de arancel"
    Incremetar Producto Base en 0.2
fin
```

La ejecución de las reglas de negocio utilizando el nuevo formato es equivalente a la definición en formato técnico de las secciones anteriores.

Consideraciones

La lista de precios de comercialización queda determinado por diferentes condiciones relacionadas al producto como ser su categoría y procedencia. Las reglas definidas se pueden analizar en forma centralizada en la tabla de decisión de la Figura G.6.

	Cursos de Acción	Reglas			
		1	2	3	4
Condiciones	CountryId	¬ Uruguay	USA	Uruguay	China
	ProductCategory			Toys	Toys
Acciones	Asignar Precio	1,01	1,05	1	1,20

Figura G.6.: Tabla de decisión – Producto

Al igual que caso de uso de la sección G.2; se agregan y modifican reglas de negocio para satisfacer las nuevas restricciones que se agregan al escenario sin tener que modificar una línea de código del programa que ejecuta.

Es importante resaltar la notoria mejora de legibilidad de las reglas de negocio entre lo que es el formato técnico versus la representación en "español" utilizando el lenguaje de "*alto nivel*" de abstracción definido (objeto *dslProduct000*), logrando una definición de regla de negocio entendible para un hispanohablante.

Escenario 3 – Factura

El escenario relacionado con el registro de facturas tiene como objetivo aplicar diferentes tipos de *validaciones*, *descuentos* y *promociones* cuando se da de alta una factura; dejando abierta la puerta para variar dichas restricciones según sea necesario en tiempo de ejecución.

La Factura está representado por dos objetos de negocio denominados `Invoice` e `InvoiceLine` como detalla la Figura G.7.

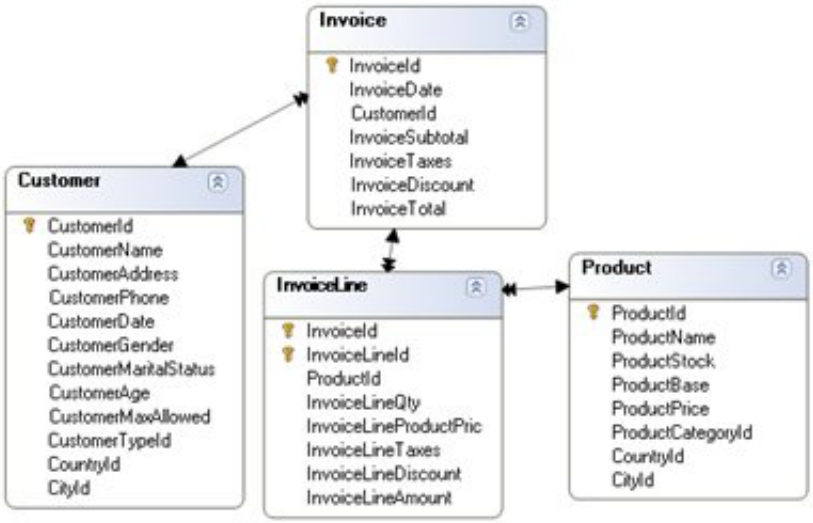


Figura G.7.: Object Factura y sus relaciones

La facturación (representada por el objeto `Invoice`) registra que un cliente realiza un pedido de una cierta cantidad de diferentes productos en un día determinado, la estructura básica es la siguiente:

- *InvoiceId*: Identificador.
- *InvoiceDate*: Fecha de generación de la factura.
- *CustomerId*: Cliente al cual se le asigna la factura.
  - *InvoiceLineId*: Item de una Factura.
  - *ProductId*: Identificador de Producto.
  - *InvoiceLineQty*: Cantidad.
  - *InvoiceLineProductPrice*: Precio asignado.
  - *InvoiceLineTaxes*: Impuesto.
  - *InvoiceLineDiscount*: Descuento.
  - *InvoiceLineAmount*: Subtotal de la línea.
- *InvoiceSubTotal*: Subtotal.
- *InvoiceTaxes*: Tasas.
- *InvoiceDiscount*: Descuento general.
- *InvoiceTotal*: Total.

Reglas

Se desea poder aplicar diferentes tipos de reglas de negocio:

- *Validaciones*: algunas restricciones que se deben de cumplir al generar una factura.
  - Un Cliente menor a 21 años no puede realizar compras.
  - Un Cliente de tipo *B* no puede realizar una Factura que exceda los $1000.
  - Un Cliente de tipo *A* no puede ordenar un producto de categoría Juguetes.
- *Descuentos*: realizar descuentos por cantidad\volumen para favorecer las economías de escala.
  - Más de 5 productos del mismo tipo llevan uno gratis.
  - Más de 4 productos diferentes, bonificar con el 5 % de descuento.
- *Promociones*: beneficiarse de ofertas existentes.

- Todos los martes, los productos de procedencia uruguaya no facturan Iva.
- 10 % de descuento para clientes de Montevideo para Enero del 2009.
- 4 items del producto 4 y 5 items del producto 5, lleva gratis 1 item del producto 6 en caso que no exista.

El formato técnico de reglas se detalla a continuación:

```
rule "Un Cliente menor a 21 años no puede realizar compras"
  no-loop true
  agenda-group "validate"
  when
    &c : Customer( Customerage < 21 )
    &i : Invoice( Customerid == &c.Customerid)
  then
    RuleEngine.AddError(1,"Menores de 21 no pueden realizar compras")
    RuleEngine.Halt()
end

rule "Un Cliente de tipo B no puede realizar una Factura que exceda los
$1000."
  no-loop true
  agenda-group "validate"
  when
    &c2 : Customer( Customertypeid == 2 ) // "B"
    &i2 : Invoice( Customerid == &c2.Customerid ,
                   Invoicesubtotal > 999 )
  then
    Msg( "validate - -Un Cliente de tipo B no puede realizar una Factura
que exceda los $1000:"+ &i2.Invoicesubtotal)
    RuleEngine.Halt()
end

rule "Un Cliente de tipo A no puede ordenar Juguetes"
  no-loop true
  agenda-group "validate"
  when
    &c3 : Customer( Customertypeid == 1 ) // "A"
    &i3 : Invoice( Customerid == &c3.Customerid )
    &il3: Invoice.Line( Productcategorydescription == "Toys" )
  then
    Msg("Un Cliente de tipo A no puede ordenar un Juguete")
    RuleEngine.Halt()
end

rule "Más de 5 productos del mismo tipo llevan uno gratis"
  no-loop true
  agenda-group "discounts"
  when
    &i4 : Invoice( )
    &il4: Invoice.Line( &qty : Invoicelineqty > 5 )
```

```
  then
    Msg("Mas de 5 productos del mismo tipo, regalar uno")
    &il4.InvoiceLineAmount = &il4.InvoiceLineAmount - &il4.InvoiceLineProductPrice
end

rule "Más de 4 productos diferentes, bonificar con el 5% de descuento"
  no-loop true
  agenda-group "discounts"
  when
    &i5 : Invoice( InvoiceLine.Count > 4 )
  then
    Msg("Mas de 4 productos diferentes, bonificar con el 5% de descuento")
    &i5.Invoicediscount = &i5.Invoicesubtotal * 0.05
end

rule "Todos los martes, los Productos uruguayos no facturan Iva"
  no-loop true
  agenda-group "promotions"
  when
    &i6 : Invoice( &myDate : Invoicedate.Day == 2 ) // Tuesday
    &p6 : Product( Countryid == 1 ) // Uruguay
    &il6 : Invoice.Line( Productid == &p6.Productid )
  then
    Msg("Todos los martes, los Productos uruguayos no facturan Iva")
    &il6.InvoiceLineTaxes = 0.00
end

rule "10% de descuento para clientes de Montevideo Enero\2009"
  no-loop true
  agenda-group "promotions"
  when
    // "Montevideo" es (1,1)
    &c7 : Customer(Countryid == 1 , Cityid == 1 )
    &i7 : Invoice( Customerid == &c7.Customerid,
                   &date : Invoicedate >= "01-Jan-2009" ,
                   Invoicedate <= "31-Jan-2009" )
  then
    Msg("10% de descuento para clientes de Montevideo 2009-01:"+&date )
    &i7.Invoicediscount = &i7.Invoicesubtotal * 0.1
end

rule "4 items del producto 4 o 5, lleva 1 gratis del producto 6"
  no-loop true
  agenda-group "promotions"
  when
    &i8 : Invoice( )
    &il8 : Invoice.Line()
    not ( &p : Product( Productid == 6 )
    and Invoice.Line( Productid == &p.Productid ) )
    exists ( &p : Product( Productid >= 4 ,
                           Productid <= 5 )
    and Invoice.Line( Productid == &p.Productid ,
```

```
                    Invoicelineqty >= 4) )
  then
    Msg("4 items del producto 4 o 5 , lleva gratis 1 item del producto 6
en caso que no exista")
    &itemInvoiceLine8 = new Invoice.Line()
    &itemInvoiceLine8.Invoicelineid = 100
    &itemInvoiceLine8.Productid=6
    &itemInvoiceLine8.Invoicelineqty=1
    &itemInvoiceLine8.Invoicelineproductprice= 1.00
    &itemInvoiceLine8.Invoicelinetaxes =0.01
    &itemInvoiceLine8.Invoicelinediscount = 0.01
    &itemInvoiceLine8.Invoicelineamount=1.00
    insert( &itemInvoiceLine8)
    &i8.Line.add( &itemInvoiceLine8 , 0 )
    update(&i8)
end
```

Para lograr la agrupación lógica de reglas de negocio en `validaciones`, `descuentos` y `promociones` se utiliza la propiedad de las reglas "`agenda-group`" que permite segmentar un juego de reglas en diferentes grupos. Notar que todas las reglas detalladas anteriormente utilizan la propiedad.

Además, para lograr el orden de evaluación deseado se agrega la regla de nombre "`boostrap`".

```
rule "boostrap"
  no-loop true
  salience 100
  when
    // nothing
  then
    Msg( "Rule engine is working...")
    // cargar el stack de "agendas" a evaluar
    RuleEngine.Focus( "discounts" )
    RuleEngine.Focus( "promotions" )
    RuleEngine.Focus( "validate" )
end
```

La regla actúa como una *metaregla* ya que en su sección de acciones está utilizando la función `Focus` del motor de evaluación para poder configurar el orden de evaluación de reglas deseado; notar que se utiliza la propiedad `salience` para aumentar la prioridad de la regla para lograr que la misma se ejecute primero y de esta manera influenciar en el orden de evaluación de las demás reglas.

Evaluación

Se crea un procedimiento que da de alta 5 facturas con información predefinida con los siguientes datos iniciales:

Atributos	**Caso 1**		**Caso 2**		**Caso 3**	**Caso 4**				**Caso 5**				
InvoiceId	610		611		612	613				614				
InvoiceDate	05/10/09		31/12/09		12/01/09	01/02/09				31/03/09				
CustomerId	1		2		3	4				5				
InvoiceLineId	1	2	1	2	1	1	2	3	4	1	2	3	4	5
ProductId	1	2	1	2	1	1	2	3	4	1	2	3	4	5
ProductCategoryId	2	2	2	2	2	2	2	2	2	2	2	2	2	2
ProductBase	100	200	100	200	100	100	200	50	100	100	200	50	100	200
InvoiceLineQty	1	2	1	2	1	1	2	3	4	1	2	3	4	5
InvoiceLineProductPrice	100	200	100	200	100	100	200	50	100	100	200	50	100	200
InvoiceLineTaxes	1	2	1	2	1	1	2	3	4	1	2	3	4	5
InvoiceLineDiscount	1	2	1	2	1	1	2	3	4	1	2	3	4	5
InvoiceLineAmount	100	400	100	400	100	100	400	150	400	100	400	150	400	1000
InvoiceSubtotal	100		1000		200	2100				4100				
InvoiceTaxes	6		6		2	20				29				
InvoiceDiscount	6		6		2	20				30				
InvoiceTotal	1000		1000		200	2100				4099				

Las reglas ejecutadas según los datos ingresados son las siguientes:

Factura	**Reglas ejecutadas**
1	Menores de 21 no pueden realizar compras
2	
3	
4	
5	Todos los martes, los Productos de procedencia uruguaya no facturan Iva Mas de 4 productos diferentes, bonificar con el 5 % de descuento

Para la primera factura se realiza una validación por lo que se cancela la evaluación de las demás reglas (ejecución del método `Halt` del motor de evaluación) y vuelve al programa llamador.

De la segunda a la cuarta factura no se registran reglas dinámicas ejecutadas por lo que los objetos de negocio permanecen incambiados; mientras que la quinta factura registra 2 reglas ejecutadas; se asigna Tasa 0 % para los productos de procedencia uruguaya impactando en la primer línea de factura:

```
<Line>
  <Invoice.Line xmlns="brsamplexev101">
  <InvoiceLineId>1</InvoiceLineId>
    <ProductId>1</ProductId>
    <ProductCategoryId>2</ProductCategoryId>
    <ProductCategoryDescription>Toys</ProductCategoryDescription>
    <ProductBase>100.00</ProductBase>
    <InvoiceLineQty>1</InvoiceLineQty>
    <InvoiceLineProductPrice>100.00</InvoiceLineProductPrice>
    <InvoiceLineTaxes>0.00</InvoiceLineTaxes>
```

```
    <InvoiceLineDiscount>1.00</InvoiceLineDiscount>
    <InvoiceLineAmount>100.00</InvoiceLineAmount>
  </Invoice.Line>
.........
```

y se bonifica con el 5 % de descuento para aquellas líneas de factura con más de 4 unidades del mismo producto.

Lenguaje de Alto nivel

Se desarrolla un lenguaje para definir las reglas de negocio en formato "español" que extiende el lenguaje utilizado en el caso de uso relacionado con el producto. Se define un objeto *RuleDSL* de nombre *dslInvoice00* con las expresiones válidas del lenguaje y su traducción al lenguaje técnico de reglas.

El juego de reglas detallado anteriormente se puede escribir de la siguiente manera:

```
expander dslInvoice000

regla "boostrap"
  no reevaluar
  prioridad 100
  si
    // nothing
  entonces
    Mostrar Mensaje "Rule engine is working..."
    // cargar los grupos de reglas a evaluar
    // notar que es una estructura LIFO
    Analizar Grupo de reglas "discounts"
    Analizar Grupo de reglas "promotions"
    Analizar Grupo de reglas "validate"
fin

regla "Un Cliente menor a 18 años no puede realizar compras"
  no reevaluar
  Agrupar en "validate"
  si
    Cliente
    - Edad < 18
    Factura
    - del Cliente
  entonces
    Agregar un Error "Menores de 18 no pueden realizar compras"
    Suspender Ejecución
fin

regla "Un Cliente de tipo B no puede realizar una Factura que exceda los
$1000."
  no reevaluar
  deshabilitar
  Agrupar en "validate"
```

```
  si
    Cliente
    - Tipo 2
    Factura
    - del Cliente
    - Subtotal > 1100
  entonces
    Mostrar Mensaje "Un Cliente de tipo B no puede realizar una Factura
que exceda los $1000"
    Suspender Ejecución
fin

regla "Un Cliente de tipo A no puede ordenar Juguetes"
  no reevaluar
  Agrupar en "validate"
  si
    Cliente
    - Tipo 1
    Factura
    - del Cliente
    Línea de Factura
    - Juguete
  entonces
    Mostrar Mensaje "Un Cliente de tipo A no puede ordenar un Juguete"
    Suspender Ejecución
fin

regla "Más de 5 productos del mismo tipo llevan uno gratis"
  no reevaluar
  Agrupar en "discounts"
  si
    Factura
    Línea de Factura
    - Cantidad > 5
  entonces
    Mostrar Mensaje "Mas de 5 productos del mismo tipo llevan uno gratis"
    Uno Va Gratis
fin

regla "Más de 4 productos diferentes, bonificar con el 5% de descuento"
  no reevaluar
  Agrupar en "discounts"
  si
    Factura
    - Cantidad de Líneas de Facura > 2
  entonces
    Mostrar Mensaje "Mas de 4 productos diferentes, bonificar con el 5%
de descuento"
    Acumular Descuento 0.05
  fin

regla "Todos los martes, los Productos de procedencia uruguaya no facturan
```

```
Iva"
  no reevaluar
  Agrupar en "promotions"
  si
    Factura
    - Martes
    Producto
    - De Uruguay
    Línea de Factura
    - mismo Producto
  entonces
    Mostrar Mensaje "Todos los martes, los Productos de procedencia uruguaya
no facturan Iva"
    Inicializar Taza en 0.00
fin

regla "10% de descuento para clientes de Montevideo Enero\2009"
  no reevaluar
  Agrupar en "promotions"
  si
    Cliente
    - De Montevideo
    Factura
    - del Cliente
    - Fecha entre "01-Jan-2009" y "31-Jan-2009"
  entonces
    Mostrar Mensaje "10% de descuento para clientes de Montevideo 2009-01"
    Asignar Descuento 0.1
fin

regla "4 items del producto 4 o 5 , lleva gratis 1 item del producto 6 en
caso que no exista"
  no reevaluar
  Agrupar en "promotions"
  si
    Factura
    No (
      Producto
      - ProductId 6
      y
      Línea de Factura
      - mismo Producto )
    Existe (
      Producto
      - Product Identifier desde 4 hasta 5
      y
      Línea de Factura
      - mismo Producto
      - Cantidad >= 4 )
  entonces
    Mostrar Mensaje "4 items del producto 4 o 5, lleva gratis 1 item del
producto 6 en caso que no exista"
```

```
Nueva línea de Factura
Asignar Línea Id 100
Asignar Producto Id 6
Asignar Cantidad 1
Asignar Base 1.00
Asignar Taza 0.01
Asignar Dto 0.01
Asignar Total 1.00
Actualizar Contexto
Agregar Línea a Factura fin
```

La ejecución del caso de uso utilizando el lenguaje de alto nivel definido es equivalente a la ejecución utilizando el lenguaje técnico de reglas.

Consideraciones

El escenario de ingreso de facturas es más representativo que los anteriores dado que intervienen varios objetos en la evaluación de reglas.

La definición de reglas tiene en cuenta varios elementos relacionados a la factura como es el cliente, las líneas de factura y los productos involucrados en dicha facturación y además se utilizan nuevos recursos del motor de evaluación y lenguaje de reglas como ser `prioridad` para influir en el orden de ejecución de reglas, `agenda-group` para agrupar reglas que se deseen evaluar por bloques.

Se suspende la ejecución del motor de reglas en la sección de consecuencias cuando se encuentra un error en los datos mediante la operación `Halt` y utiliza operadores lógicos como por ejemplo `not` y el cuantificador universal `exists`.

Notar que todas estas características se ocultan utilizando el lenguaje de reglas de alto nivel definido.

Escenario 4 – Workflow

El escenario 4 explora el uso de reglas de negocio integrado a los procesos de negocio, de manera de lograr cierto control sobre lógica de negocio que influya en los flujos de procesos.

Supongamos un proceso de negocio como se detalla en la Figura G.8 donde se generan pedidos que deben ser autorizados por un administrativo y es necesario toman algunas decisiones de cómo se sigue el flujo de proceso.

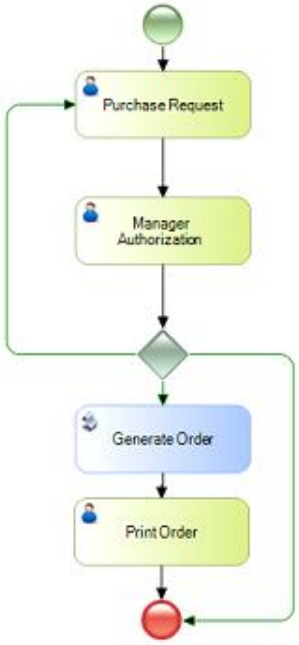


Figura G.8.: Proceso de Negocio de ejemplo

En este caso en el condicional posterior a la tarea *Manager Authorization* interesa tener la capacidad de evaluar reglas de negocio para poder realizar fácilmente modificaciones en caso que sea necesario. Inicialmente las reglas que rigen al condicional son las siguientes:

- *Regla 1*: Rechazar Orden de compra si la misma no está completa, por ejemplo tiene una descripción de pedido vacía.

```
rule "Rechazar"
  no-loop true
  salience 10
  when
    &myRequest:Request(ReqDsc=="")
  then
    Msg("Rechazar...")
    &myRequest.ReqAut = 2
    RuleEngine.Halt()
end
```

- *Regla 2*: Generar Orden de Compra si el total es mayor a 20.

```
rule "Generar Orden
  no-loop true
  when
    &myRequest:Request(&tot:ReqTotAmt > 20)
  then
```

```
    Msg("Generar Orden...")
    &myRequest.ReqAut = 1
    RuleEngine.Halt()
end
```

- *Regla 3*: Por defecto finalizar el flujo de ejecución.

```
rule "Finalizar"
  no-loop true
  salience -10
  when
    &myRequest:Request()
  then
    Msg("Finalizar...")
    &myRequest.ReqAut = 3
end
```

Evaluación

La evaluación del escenario se realiza en forma manual, ingresando inicialmente un requerimiento sin definir su descripción; se ejecuta la regla "Rechazar" al llegar al condicional correspondiente por lo que se ejecuta la decisión de volver al inicio del proceso a la tarea "*Purchase Request*".

Posteriormente se actualiza el requerimiento agregando varias líneas de manera de lograr que el total sea mayor a 20. En este caso al evaluarse la decisión a través de la ejecución de reglas se ejecuta la regla "*Generar Orden*" siguiendo el flujo el proceso "*Generate Orden*" y posteriormente finaliza el flujo normal de ejecución de la instancia de proceso.

Lenguaje de Alto nivel

Para las tres reglas detalladas anteriormente se desarrollan dos objetos *RuleDSL* para hacer pruebas con dos lenguajes diferentes, por un lado uno aproximado al lenguaje inglés y el segundo una aproximación al lenguaje japonés.

El objeto *dslFlowEnglish* que centraliza el lenguaje en "formato inglés" para manipular el proceso de negocio de ejemplo se detalla en la sección F.2 del Anexo F. Las reglas se reescriben de la siguiente forma:

```
expander dslFlowEnglish
rule "Reject"
 Do not loop
 when
  Request Description is Empty
 then
  Message "Reject..."
  Set Authorization Status "2"
end

rule "Generate Order"
 Do not loop
 when
  A Request
   - Total greater than 20
 then
  Message "Generate Order..."
  Set Authorization Status "1"
  Stop Execution
end

rule "GoToEnd"
 Do not loop
 Low Priority
 when
  A Request
 then
  Message "GoToEnd"
  Set Authorization Status "3"
end
```

Figura G.9.: Reglas utilizando parametrización al inglés

La ejecución del caso de uso utilizando las nuevas expresiones del objeto *RuleDSL* es equivalente.

Al igual que con el escenario relacionado al Cliente en donde se experimenta con un lenguaje de reglas de alto nivel para hispanohablantes, en este caso se define además un nuevo objeto *dslFlowJapanese* donde se centraliza un lenguaje de reglas de ejemplo de alto nivel de abstracción para ser utilizado en el mercado japonés (sección F.2 del Anexo F):

```
expander dslFlowJapanese
rule "Reject"
 輪にならないでください
 when
  要請説明がむなしいです
 then
  メッセージ "Reject"
  セットされた委任ステータス "2"
end

rule "Generate Order"
 輪にならないでください
 when
  要請
   - より素晴らしいことを合計してください 20
 then
  メッセージ "Generate Order"
  セットされた委任ステータス "1"
end

rule "Ending"
 輪にならないでください
 when
  要請
 then
  メッセージ "Ending"
  セットされた委任ステータス "3"
  停止実行
end
```

Figura G.10.: Reglas utilizando parametrización al japonés

La ejecución del escenario es equivalente utilizando cualquiera de los “dialectos” (lenguajes de alto nivel) detallados anteriormente al igual que con el lenguaje técnico detallado al inicio de la sección.

Consideraciones

Las decisiones del flujo de proceso se encuentran centralizadas en tres reglas de negocio, pudiéndose modificar según se requiera.

Se utiliza a las reglas de negocio como restricciones sobre los procesos empresariales; donde no es necesario conocer a priori el detalle de cómo va a funcionar el proceso, sino que se agrega un grado de libertad al centralizar ciertas decisiones y restricciones empresariales en reglas de negocio que podrán definirse en tiempo de implantación logrando una definición de procesos de negocio desestructurada.

Además, se experimenta la definición de dichas reglas en dos lenguajes de alto nivel de abstracción diferentes con el objetivo de simular la implantación de la aplicación de ejemplo en diferentes mercados. En este caso se utiliza la misma infraestructura en todas las implantaciones (ya que se trata de la misma aplicación) y en cada contexto sólo se debe personalizar en objetos de tipo *RuleDSL* el lenguaje que maneja el experto del negocio.

Escenario 5 – OAV

El modelo de datos OAV (Objeto-Atributo-Valor) también conocido como EAV (Entidad–Atributo-Valor) permite extender la información de una base de datos sin necesidad de modificar la estructura de la misma ni realizar ningún tipo de reorganización física. En matemática a este tipo de modelo se le conoce como matriz dispersa.

Supongamos la entidad de *Clientes* (detallada anteriormente) a la cual se le desea definir nuevos atributos secundarios. El objetivo es diseñar una aplicación flexible, que se adapte a diferentes realidades. Encontramos que si bien existen un conjunto de atributos de clientes que van a ser comunes, en muchos casos son insuficientes. Además se desea brindar la posibilidad a los usuarios del sistema que agreguen nuevos atributos a la entidad, sin necesidad de modificar la aplicación (ni la base de datos, ni los programas). Este es un caso clásico de aplicación del patrón *Objeto-Atributo-Valor*.

Para diseñar una aplicación que se ajuste a estos requerimientos, diseñamos una entidad como detalla la Figura G.11.

Name	Type	Description	Is Collection
ExtendedCustomer		Extended Customer	☐
CustomerId	Attribute:CustomerId	Customer Id	☐
CustomerName	Attribute:CustomerName	Customer Name	☐
CustomerAddress	Attribute:CustomerAddress	Customer Address	☐
CustomerPhone	Attribute:CustomerPhone	Customer Phone	☐
CustomerDate	Attribute:CustomerDate	Customer Date	☐
CustomerGender	Attribute:CustomerGender	Customer Gender	☐
CustomerMaritalStatus	Attribute:CustomerMaritalStatus	Customer Marital Status	☐
CustomerAge	Attribute:CustomerAge	Customer Age	☐
CustomerMaxAllowed	Attribute:CustomerMaxAllowed	Customer Max Allowed	☐
CustomerTypeId	Attribute:CustomerTypeId	Customer Type Id	☐
CustomerTypeName	Attribute:CustomerTypeName	Customer Type Name	☐
CountryId	Attribute:CountryId	Country Id	☐
CityId	Attribute:CityId	City Id	☐
Extensions		Extensions	☑
Item		Item	
ObjectAttributeId	Attribute:ObjectAttributeId	Object Attribute Id	☐
ObjectAttributeName	Attribute:ObjectAttributeName	Object Attribute Name	☐
ObjectAttributeType	Attribute:ObjectAttributeType	Object Attribute Type	☐
ObjectAttributeValue	Attribute:ObjectAttributeValue	Object Attribute Value	☐

Figura G.11.: Cliente Extendido

Notar que el cliente (le denominamos `ExtendedCustomer`) tiene almacenada la información básica de clientes (nombre, teléfono, género etc. . .) que será aquella

información que es igual para todas las instalaciones de la aplicación y posteriormente define una lista de Items (propiedades genéricas representadas por la colección `Extensions`) que serán aquellas definiciones que podrán variar en las diferentes instalaciones de la aplicación.

Supongamos dos clientes de prueba (Adán y Eva) con tres atributos instanciados en tiempo de instalación:

- *Deporte preferido* utilizando el Identificador 1.
- *Altura* utilizando el Identificador 2.
- *Última actualización* utilizando el Identificador 3.

Según la estructura detallada anteriormente podría modelarse de la siguiente forma:

```
{
  // Adan
  CustomerId = 1
  CustomerName = "Adan"
  ......
  CityId = 1
  Extensions
  {
    Item.Insert(1,"1",AttributeType.Character,"Soccer")
    Item.Insert(1,"2",AttributeType.Numeric,"1.70")
    Item.Insert(1,"3",AttributeType.Date,"01/03/2000")
  }
}

{
  // Eva
  CustomerId = &CustomerId
  CustomerName = &CustomerName
  ......
  CityId = 1
  Extensions
  {
    Item.Insert(1,"1",AttributeType.Character,"Gym")
    Item.Insert(1,"2",AttributeType.Numeric,"1.63")
    Item.Insert(1,"3",AttributeType.Date,"12/03/2000")
  }
}

Subgroup Item( &ObjectAttributeId, &ObjectAttributeName,
               &ObjectAttributeType, &ObjectAttributeValue )
  Item
  {
    ObjectAttributeId = &ObjectAttributeId
    ObjectAttributeName = &ObjectAttributeName
    ObjectAttributeType = &ObjectAttributeType
```

```
    ObjectAttributeValue = &ObjectAttributeValue
  }
Endsubgroup
```

Utilizando el esquema de definición de reglas de negocio se desea tener la capacidad de definir reglas sobre los atributos dinámicos. Esto quiere decir que es necesario manipular la lista de `Extensiones` del objeto `ExtendedCustomer` para poder definir dichas restricciones sobre los "*atributos dinámicos*" que se definan.

Supongamos que se quiere aplicar una regla para el cliente de nombre "*Adan*" y que tenga definido el atributo dinámico "*Deporte preferido*", la especificación de la regla podría definirse como:

```
rule "Adan Found"
  when
    &c:ExtendedCustomer(&name:CustomerName=="Adan",&l:Extensions)
    &i:ExtendedCustomer.Item(ObjectAttributeName=="1") from &l
  then
    Msg("Hola Adan:"+&c.ToXML())
end
```

En la sección de condiciones, la primera restricción simplemente filtra por el nombre del cliente y proyecta en `&l` la colección de extensiones. La segunda restricción se encarga de filtrar por el identificador "1" dentro de la colección para obtener la representación interna de "`Deporte preferido`", es decir busca un objeto con ese identificador dentro de la colección de extensiones.

Se detectó que este tipo de regla no es posible definirla debido a como se manejan las colecciones de elementos desde GeneXus a nivel del código generado en la plataforma y cómo funciona el "matcheo" de patrones en el motor de reglas utilizado.

Por un lado la generación de código GeneXus al manipular colecciones de elementos la realiza con objetos genérico. Esto quiere decir que para referenciar un elemento dentro de la colección es necesario realizar un *Cast*[1] al tipo de objeto destino para poder trabajar con las propiedades y métodos de los elementos. En el contexto de las reglas dinámicas, donde se desea definir restricciones sobre atributos dinámicos implica conocer el tipo del objeto que fue almacenado en la colección para poder resolver la correspondiente conversión y realizar el filtro deseado.

Es decir, supongamos el siguiente código GeneXus que define un atributo dinámico y lo agrega a una colección:

```
&Extension01 = new()
&Extension01.ObjectAttributeId = 1
&Extension01.ObjectAttributeName = "1"
&Extension01.ObjectAttributeType = AttributeType.Character
```

[1] `http://en.wikipedia.org/wiki/Type_conversion`

```
&Extension01.ObjectAttributeValue = "Soccer"
&ExtendedCustomer.Extensions.Add(&Extension01)
&Extension01 = &ExtendedCustomer.Extensions.Item(1)
```

siendo `&ExtendedCustomer` de tipo `ExtendedCustomer` y `&Extensions01` de tipo `ExtendedCustomer.Item`. La generación de código para la plataforma Java es la siguiente:

```
AV65Extension01 = (SdtExtendedCustomer_Item)new SdtExtendedCustomer_Item(remoteHandle,
context);
AV65Extension01.setgxTv_SdtExtendedCustomer_Item_Objectattributeid(1);
AV65Extension01.setgxTv_SdtExtendedCustomer_Item_Objectattributename("1");
AV65Extension01.setgxTv_SdtExtendedCustomer_Item_Objectattributetype(0);
AV65Extension01.setgxTv_SdtExtendedCustomer_Item_Objectattributevalue("Soccer");
AV64ExtendedCustomer.getgxTv_SdtExtendedCustomer_Extensions().add(AV65Extension01,
0);
AV65Extension01 = (SdtExtendedCustomer_Item)
 (
  (SdtExtendedCustomer_Item)
   AV64ExtendedCustomer.getgxTv_SdtExtendedCustomer_Extensions().elementAt(-1+1)
 );
```

Notar que para poder referenciar un elemento de tipo `ExtendedCustomer.Item` es necesario realizar un *cast* una vez que se obtiene el elemento de la colección correspondiente (detallado en la última línea de la sección de código anterior); que se corresponde con el código GeneXus:

```
&Extension01 = &ExtendedCustomer.Extensions.Item(1)
```

La herramienta utilizada (Drools 5.0) no permite definir casts explícitos en las condiciones[2] de las reglas por lo que no se permite condicionar reglas que involucren estos atributos dinámicos.

Para solucionar el problema en forma temporal y validar el escenario de uso se crea un objeto "*Helper*" GeneXus denominado `HasExtendedProperty`, que resuelve con código GeneXus lo que no puede resolverse desde la plataforma.

El programa `HasExtendedProperty` recibe como argumento un objeto de tipo `ExtendedCustomer` y un nombre de propiedad dinámica retornando un tipo de dato *Boolean* (verdadero o falso) indicando si la propiedad existe en la colección de "atributos dinámicos" del cliente. Por lo que para definir una condición de una regla sobre un atributo dinámico se utiliza un procedimiento GeneXus como si fuera una función.

La firma del objeto "`HasExtendedProperty`" es la siguiente:

```
Parm(in:&ExtendedCustomer,in:&ObjectAttributeName,out:&result);
```

[2] `http://osdir.com/ml/java.drools.user/2007-01/msg00204.html`

y ejecuta el siguiente código GeneXus:

```
&result = false
For &Property in &ExtendedCustomer.Extensions
  If (&Property.ObjectAttributeName = &ObjectAttributeName)
    &result = true
    return
  endif
endfor
```

La regla de negocio se puede definir de la siguiente manera, utilizando el nuevo objeto `HasExtendedProperty`; se tiene que utilizar la función `eval` para procesar el resultado de la evaluación del procedimiento en la sección de condiciones, es decir:

```
rule "Adan Found"
  when
    &c:ExtendedCustomer(&name:CustomerName=="Adan",&l:Extensions)
    eval(HasExtendedProperty(&c,"1"))
  then
    Msg("Hola Adan:"+&c.ToXML())
End
```

Debido a restricciones a nivel de la gramática en el lenguaje de reglas GeneXus definido (ver Anexo F) no es posible salvar dicha regla debido a que la función `eval` no fue implementada en su totalidad. La restricción no permite que la función `eval` reciba cualquier tipo de expresión, como es el caso de una llamada a un objeto con varios argumentos retornando un valor booleano.

Igualmente para validar el escenario se realizan pruebas sobre el lenguaje de definición de reglas *DRL* directamente en la plataforma de ejecución, pudiéndose definir la regla de la siguiente forma:

```
rule "Adan Found"
  when
    $c:com.sample.SdtExtendedCustomer(
        $name:gxTv_SdtExtendedCustomer_Customername=="Adan",
        $l:gxTv_SdtExtendedCustomer_Extensions )
    eval(new com.sample.hasextendedproperty(-1).executeUdp($c,"1"))
  then
    com.ruleset.api.MessageHelper.msg(
      "Hola Adan:"+
      $c.toxml(false,"ExtendedCustomer","brsamplexev101")
    );
end
```

Utilizando el juego de datos detallado al inicio de la sección se ejecuta correctamente la regla sobre el *cliente extendido* "Adan" con el atributo dinámico `Deporte preferido`.

```
Hola Adan:<ExtendedCustomer xmlns="brsamplexev101">
  <CustomerId>1</CustomerId>
  <CustomerName>Adan</CustomerName>
  <CustomerAddress>Adan</CustomerAddress>
  <CustomerPhone>Adan</CustomerPhone>
  ........................
  <Extensions>
    <Item>
      <ObjectAttributeId>1</ObjectAttributeId>
      <ObjectAttributeName>1</ObjectAttributeName>
      <ObjectAttributeType>0</ObjectAttributeType>
      <ObjectAttributeValue>Soccer</ObjectAttributeValue>
    </Item>
    <Item>
      <ObjectAttributeId>1</ObjectAttributeId>
      <ObjectAttributeName>2</ObjectAttributeName>
      <ObjectAttributeType>1</ObjectAttributeType>
      <ObjectAttributeValue>1.70</ObjectAttributeValue>
    </Item>
    <Item>
      <ObjectAttributeId>1</ObjectAttributeId>
      <ObjectAttributeName>3</ObjectAttributeName>
      <ObjectAttributeType>2</ObjectAttributeType>
      <ObjectAttributeValue>01/03/2000</ObjectAttributeValue>
    </Item>
  </Extensions>
</ExtendedCustomer>
```

Consideraciones

El caso de uso OAV no se pudo completar en su totalidad utilizando código GeneXus. Inicialmente se detectó una restricción en la forma de manipular las colecciones de objetos en el motor de reglas utilizado en relación a la generación de código GeneXus para la plataforma Java. Se decide utilizar código GeneXus para poder solucionar el caso y poder definir condiciones de una regla que involucren colecciones de objetos.

Se encuentra una nueva restricción, en este caso en la definición de la gramática del lenguaje de reglas diseñado ("GXRL") donde no se soporta la evaluación de condiciones complejas a partir de las producciones que dependan de la función *eval*.

Sin embargo, manipulando directamente el lenguaje de reglas y código generado GeneXus se pudo ejecutar la regla de negocio. Por lo tanto, es viable resolver este tipo de escenario realizando modificaciones a nivel de código generado o agregando nuevos artefactos GeneXus para poder solventar el problema desde el lenguaje de reglas diseñado. Se utilizó dicho enfoque para finalizar el escenario agregando "Helper Objects[3]" en GeneXus, los cuales se utilizan desde las condiciones de reglas con el objetivo de manipular colecciones de objetos.

[3] `http://forums.asp.net/t/488434.aspx`

Escenario 6 – Interacción con código GeneXus

Un recurso muy útil que es importante en el contexto de ejecución de reglas, es tener la posibilidad de ejecutar procedimientos GeneXus para reutilizar lógica de negocio ya existente.

Este escenario plantea la investigación de cómo lograr ejecución de objetos GeneXus desde la consecuencia de reglas. Para ello se reutiliza parte del caso de uso de facturación, especificando el siguiente juego de reglas a utilizarse con el mismo juego de datos.

```
rule "Invoice02 - Call procedure"
  when
  &i8 : Invoice(
      &CustomerId:CustomerId==1 ,
      &InvoiceAmount:InvoiceSubTotal ,
      &InvoiceDate:InvoiceDate ,
      &CustomerName:CustomerName)
  then
    Msg("Invoice02 - Call procedure")
    if (1==1)
      // Modifico el cliente predeterminado
      &i8.CustomerId = 2
    else
      &i8.CustomerId = 1
    endif
    // Creo una linea de factura
    &itemInvoiceLine8 = new Invoice.Line()
    &itemInvoiceLine8.InvoiceLineId = 100
    &itemInvoiceLine8.ProductId = 6
    &itemInvoiceLine8.InvoiceLineQty = 1
    &itemInvoiceLine8.InvoiceLineProductPrice= 1.00
    &itemInvoiceLine8.InvoiceLineTaxes = 0.01
    &itemInvoiceLine8.InvoiceLineDiscount = 0.01
    &itemInvoiceLine8.InvoiceLineAmount = 1.00
    // Llamo a un objeto GX
    gxProcedure01.Call(&i8, &itemInvoiceLine8, &CustomerId, &InvoiceAmount
,&CustomerName, true)
    gxLog.Call(&i8.InvoiceId,"Invoice02 - Call procedure")
end
```

La sección de acciones de la regla entre otras cosas utiliza la construcción `If...then...else`, creación de objetos de negocio como es el caso del constructor "`new Invoice.Line()`", asignación de propiedades y finalmente ejecución de procedimientos GeneXus como son los objetos `gxLog` y `gxProcedure01`.

Evaluación

Reutilizando el juego de datos del escenario de facturación se obtiene el siguiente resultado:

```
Invoice02 - Call procedure
gxProcedure01 is being executed.
Insert OK
<Invoice xmlns="brsamplexev101">
  <InvoiceId>620</InvoiceId>
  <InvoiceDate>2009-10-08</InvoiceDate>
  <CustomerId>2</CustomerId>
  <CustomerName>Customer 0</CustomerName>
  <Line>
    <Invoice.Line xmlns="brsamplexev101">
      <InvoiceLineId>1</InvoiceLineId>
      ..................
```

Interesa resaltar que el mensaje "gxProcedure01 is being executed" se llama desde el código del objeto gxProcedure01, que contiene el siguiente código GeneXus:

```
&msg = &Pgmname.Trim() + " is being executed."
msg(&msg,status)
```

El objeto gxLog también fue ejecutado ya que se insertó un registro en una tabla de log genérica tal como detalla el código procedural del objeto correspondiente:

```
&LogDsc = Format("%1:%2",&id,&message)
&date = servernow()
// Log Message
New
  LogDsc = &LogDsc
  LogDate = &date
  When Duplicate
  // Do nothing, is an autonumber
EndNew
```

Listando la tabla de log se evidencia el siguiente registro resultado de la ejecución:

ID	Description	Timestamp
28	625:Invoice02 – Call procedure	01/12/2009 09:00 AM

La columna *ID* es un atributo autonumber, la columna *Description* se corresponde con el atributo *LogDsc* y la columna *Timestamp* se corresponde con el atributo *LogDate*.

Consideraciones

Se logra ejecutar en forma transparente objetos GeneXus de tipo procedimiento; solo se implementa el pasaje de parámetros de tipo *In*. Queda fuera del escenario realizar pruebas de ejecución sobre objetos de tipo DataProvider, ya que en este caso el pasaje de parámetros es *InOut* en forma predeterminada.

Escenario 7 – Modelo Universal de Datos

Los modelos universales de datos (UDM - Universal Data Model) son modelos de datos que apuntan a almacenar cualquier tipo de información sin necesidad de modificar el esquema de la base de datos.

Todos estos modelos de datos tienen un enfoque común respecto a como descomponen la información, generalmente en:

- *Entidades*: representan objetos del mundo real como por ejemplo: Cliente, Persona, Factura, Producto.
- *Atributos*: Las entidades tienen "atributos". En general dos objetos de la misma *Entidad* tendrán atributos más o menos parecidos, el valor de los atributos es lo que diferencia un objeto de otro.
- *Relación entre Entidades*: Cada *Entidad* no está aislada de su contexto se relaciona de diferente forma con otras entidades. Una *Factura* está relacionada con el *Cliente* y *Producto*, a su vez el *Producto* está relacionado con *Categoría de Producto* y así sucesivamente.

El escenario OAV resuelve el problema de extender las propiedades de cualquier entidad de un dominio, sin embargo no resuelve escenarios en donde se desea agregar y relacionar nuevas entidades; es decir poder instanciar "Entidades" que no fueron previstas inicialmente en el sistema; pudiendo relacionarlas con entidades ya existentes. A este tipo de modelado flexible se le denomina modelo universal de datos (UDM - Universal Data Model).

Si se modifica el modelo para el tratamiento del caso OAV se puede obtener un modelo extensible genérico de manejo de entidades; supongamos definimos un objeto para manipular Entidades que le llamamos "*EntiySDT*" detallado en la Figura G.12.

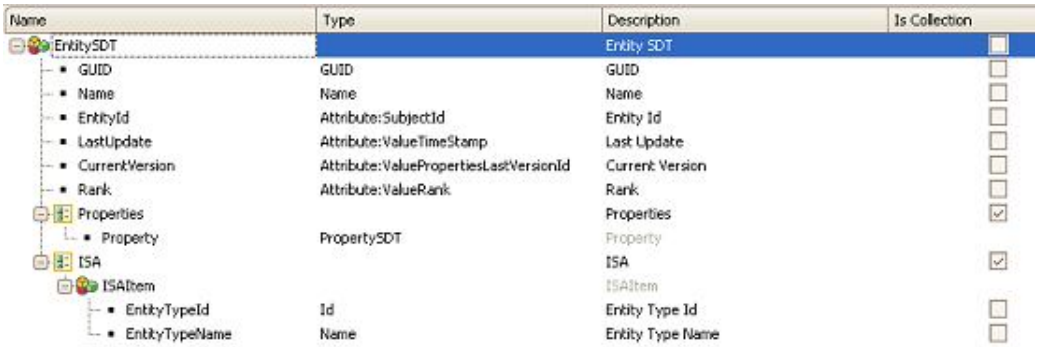


Figura G.12.: Entidad genérica EntitySDT

Define en términos generales información básica, un puntero a otros objetos que describen la entidad (atributo *EntityID*) y una lista de propiedades complejas

(propiedades compuestas por otras propiedades) representada por las colecciones *Properties* e *ISA*.

Supongamos que se tiene la entidad "`Lenovo T60`" de tipo "`Notebook`", con sus propiedades `Memoria`, `Pantalla` etc.... Una representación del objeto con el modelo anterior es la siguiente:

```
EntitySDT
{
  Name = 'Lenovo T60'
  IsA.Insert('Notebook')
  Properties
  {
    AProperty.Insert('Memoria', ValueTypeDomain.Literal, '2 GB')
    Property
    {
      PropertyId = GetPropertyId('Almacenamiento')
      Properties
      {
        Property
        {
          PropertyId = GetPropertyId('Type')
          PropertyValueId = GetSetValue(ValueTypeDomain.Literal, 'Disco')
          Properties
          {
            AProperty.Insert('RPM',ValueTypeDomain.Number,'5500')
            AProperty.Insert('Size',ValueTypeDomain.Literal,'120GB')
          }
        }
      }
    }
    Property
    {
      PropertyId = GetPropertyId('Pantalla')
      Properties
      {
        AProperty.Insert('Type',ValueTypeDomain.Literal,'LCD')
        AProperty.Insert('Size',ValueTypeDomain.Literal,'15"')
      }
    }
  }
}
```

Interesa poder definir reglas sobre cualquier entidad para poder agregar la lógica de negocio correspondiente.

Supongamos que se quiere aplicar una regla para el objeto anterior tal que las entidades "`Lenovo`" tiene una promoción especial con entrega inmediata; se podría especificar una regla que aplique sobre entidades "`Lenovo T60`" y agregue una nueva propiedad "`Entrega Inmediata`" y una nueva relación "`Oferta Especial`" para categorizar la oferta; la especificación de la regla podría definirse como:

```
rule "Ejemplo Lenovo"
  when
  &e : EntitySDT( &n:Name=="Lenovo T60", &p:Properties, &id:EntityId,
                  &r:Rank, &is:ISA )
  then
    Msg("Agregando una Relacion y Propiedad a " + &n)
    // creo una relación nueva "Oferta Especial"
    &entitySDTIsA01 = new EntitySDT.ISAItem()
    &entitySDTIsA01.EntityTypeId = 1
    &entitySDTIsA01.EntityTypeName = "Oferta Especial"
    &e.ISA.Add(&entitySDTIsA01)
    // Creo una propiedad nueva "Entrega Inmediata"
    &PropertySDTitem01 = new PropertySDT()
    &PropertySDTitem01.PropertyId = 1
    &PropertySDTitem01.PropertyName = "Entrega Inmediata"
    &PropertySDTitem01.PropertyValueId = 1
    &PropertySDTitem01.PropertyValueName = "Entrega Inmediata"
    &PropertySDTitem01.PropertyValueData = "Entrega Inmediata"
    &e.Properties.Add(&PropertySDTitem01)
    //Muestro resultado
    Msg(&e.ToXML())
end
```

La sección de condiciones de la regla aplica una restricción para filtrar la entidad deseada (`Lenovo T60`) y proyecta diferentes propiedades de la entidad; posteriormente en la sección de acciones se manipulan colecciones y propiedades del objeto proyectado obteniéndose el siguiente resultado:

```
Agregando una Relacion y Propiedad a Lenovo T60
<EntitySDT xmlns="brsamplexev101">
  <GUID>00000000-0000-0000-0000-000000000000</GUID>
  <Name>Lenovo T60</Name>
  <EntityId>0</EntityId>
  ............
  <Properties xmlns="Permatron">
    ..................
    <Property xmlns="Permatron">
      <PropertyId>1</PropertyId>
      <PropertyName>Entrega Inmediata</PropertyName>
      ......................
      </Property>
    </Properties>
    <ISA xmlns="Permatron">
      <ISAItem xmlns="Permatron">
        <EntityTypeId>0</EntityTypeId>
        <EntityTypeName/>
      </ISAItem>
      <ISAItem xmlns="Permatron">
        <EntityTypeId>1</EntityTypeId>
        <EntityTypeName>Oferta Especial</EntityTypeName>
      </ISAItem>
```

```
    </ISA>
</EntitySDT>
```

Notar que en la sección de propiedades del objeto proyectado se agrega la propiedad "`Entrega Inmediata`" y en la sección de relaciones representado por la colección *ISAItem* se agrega "`Oferta Especial`".

Consideraciones

Al igual que el escenario OAV, el caso de uso no se pudo completar en su totalidad con código GeneXus dado que "sufre" de los mismos problemas encontrados en el escenario OAV; en el sentido que para poder referenciar tanto propiedades como relaciones, en las condiciones de las reglas se tiene que manipular colecciones de objetos.

Solventando las limitaciones definidas en la gramática y utilizando "*Helper Objects*" para utilizar colecciones en condiciones de reglas, es viable especificar cualquier tipo de regla sobre entidades genéricas definidas en un modelo universal de datos.

Escenario 8 – Uso de Operaciones

Existen varios métodos que se pueden utilizar por código desde GeneXus para poder informar al motor de evaluación de reglas del cambio de estado de diferentes objetos. Basados en el caso de uso *Hola Mundo*, se agregan métodos al objeto externo *RulesetEngine* para utilizar las operaciones `Update` y `Retract`.

Al ejemplo inicial se agrega el siguiente código:

```
&ruleset = "HelloWorld002"
&myMessage.Message = "Hello World"
&myMessage.Status = true // Message.HELLO
msg("startup",status)
msg(&myMessage.ToXml() ,status)
&myRulesetEngine.Ruleset(&ruleset)
&ret = &myRulesetEngine.Insert(&myMessage)
&myRulesetEngine.FireRules()
msg("Fact Insertion and Rules fired",status)
&myMessage.Status = true
&myMessage.Message = "Another message"
&myRulesetEngine.Update(&myMessage,&ret)
&myRulesetEngine.FireRules()
msg("Fact updated and Rules fired again",status)
msg(&myMessage.ToXml(),status)
&myMessage.Status = true
&myRulesetEngine.Retract(&ret)
&myRulesetEngine.FireRules()
msg("Fact retraction and Rules fired again",status)
&myRulesetEngine.Close() // close session
```

Al juego de reglas inicial:

```
rule "Hello World"
  when
    &m : MyMessage( Status == true , &theMessage : Message )
  then
    Msg( &theMessage )
    &m.Message = "Goodbye cruel world"
    &m.Status = false
    update( &m )
end

rule "GoodBye"
  when
    MyMessage( Status == false, &theMessage : Message )
  then
    Msg( &theMessage )
end
```

se agrega lo siguiente:

```
rule "No Message"
  when
    not( MyMessage() )
  then
    Msg( "No hay Mensaje" )
end
```

El resultado de la ejecución del caso de uso es:

```
\========== Execution started ==========
"C:\Archivos de programa\Java\jdk1.6.0\bin\java.exe" ahelloworld01

startup
<MyMessage xmlns="brsamplexev101">
  <Status>true</Status>
  <Message>Hello World</Message>
</MyMessage>
Hello World
Goodbye cruel world
Fact Insertion and Rules fired
Another message
Goodbye cruel world
Fact updated and Rules fired again
<MyMessage xmlns="brsamplexev101">
  <Status>false</Status>
  <Message>Goodbye cruel world</Message>
</MyMessage>
No hay Mensaje
Fact retraction and Rules fired again
Execution Success
```

Consideraciones

En este caso interesa resaltar la modificación del objeto `MyMessage` desde las consecuencias de reglas y código GeneXus informando al motor de reglas de la actualización del objeto con la operación `Update` lográndose que se ejecuten nuevamente las reglas.

Al final de la ejecución de la prueba se utiliza la operación `Retract` y al ejecutar las reglas se dispara la regla "`No hay Mensaje`" indicando que ya no existen objetos de tipo `MyMessage` en la memoria de trabajo del motor, posteriormente se cierra la sesión.

Escenario 9 – "Truth Maintenance"

Al motor de evaluación de reglas también se lo puede utilizar como mecanismo de inferencia o razonamiento. El mecanismo *Truth Maintenance* es un recurso que proveen los motores para eliminar hechos de la memoria de trabajo en forma automática cuando ciertas hipótesis ya no son válidas; generalmente implementadas mediante la operación *InsertLogical*.

Dicho operador es similar a la operación `insert` teniendo en cuenta que el objeto insertado será eliminado en forma automática de la memoria de trabajo cuando no existan más hechos que soporten la "verdad" de la regla que hizo la inserción lógica. En todo momento se cumplen las siguientes condiciones:

- El objeto es único.
- La validez del objeto es mantenida en forma automática.

Truth Maintenance es un recurso interesante para especificar dependencias lógicas entre hechos. Todos los hechos que se insertan en la sección derecha de una regla (acciones) utilizando el operador *InsertLogical*, se tornan dependientes de la sección izquierda (condiciones). En caso que alguna de las condiciones se tornen inválidas, los hechos dependientes se eliminan en forma automática.

Supongamos el siguiente ejemplo que se basa sobre el escenario de uso básico *Hola Mundo*. Se parte de los siguientes mensajes que son insertados en la memoria de trabajo del motor:

```
MyMessages
{
  MyMessage
  {
    Message = "1. Mensaje OK"
    Status = true
  }
  MyMessage
  {
```

```
    Message = "2. Mensaje FALSE"
    Status = false
  }
  MyMessage
  {
    Message = "3. Mensaje FALSE"
    Status = false
  }
}
```

A partir de tres objetos de tipo mensaje el objetivo es lograr que todos los mensajes pasen de status `false` a `true`; en este caso los últimos dos mensajes.
Para ello se utiliza el recurso inserción lógica, cuando existe un mensaje falso se inserta un objeto de tipo *Alerta*.

```
rule "Mensaje Incorrecto"
  salience 10
  when
    exists( &m:MyMessage( Status == false ) )
  then
    &vAlert = new Alert()
    insertLogical( &vAlert )
end
```

Se crea una regla que muestra un mensaje indicando que se tienen que procesar las alertas:

```
rule "Alertas"
  salience 10
  when
    exists( Alert() )
  then
    Msg("Hay Alertas a procesar")
end
```

Se crea otra regla para "arreglar los problemas", en este caso se actualiza el status de los mensajes (notar que en las condiciones se está apoyando en el objeto *Alerta* insertado en forma lógica):

```
rule "Arreglando Problemas..."
  when
    &m: MyMessage( Status == false )
    exists( Alert() )
  then
    Msg( "Arreglando " + &m.Message)
    &m.Status = true
    update(&m)
end
```

Se crea una regla que aplica cuando ya no existen alertas para procesar:

```
rule "Ya no hay Alertas"
  when
    not( Alert() )
  then
    Msg( "No hay Alertas!!!")
end
```

Es en la regla "`Mensaje Incorrecto`" en donde se declara la dependencia lógica de las condiciones. Mientras exista al menos un mensaje (objeto `MyMensaje`) con status `false` se tiene que mantener la alerta.

Con la regla "`Arreglando problemas`" se emula la situación en donde se modifican las propiedades de los mensajes para dejarlos en un status válido. Al procesar los dos mensajes con status `false` ya no se puede soportar la inserción lógica por lo que en forma automática se hace `retract` del objeto *Alerta*; esto trae como consecuencia que se disparen nuevas reglas, en este caso, "`Ya no hay Alertas`".

El resultado en ejecución es el siguiente:

```
\========== Execution started ==========
"C:\Archivos de programa\Java\jdk1.6.0\bin\java.exe" com.sample.atms02

inserting 1. Mensaje OK
inserting 2. Mensaje FALSE
inserting 3. Mensaje FALSE
Hay Alertas a procesar
Arreglando 3. Mensaje FALSE
Arreglando 2. Mensaje FALSE
No hay Alertas!!!
Execution Success
```

Consideraciones

Este recurso es muy utilizado por ejemplo para detectar transacciones con probabilidad de fraude. Se especifican reglas que ante cierta casuística simplemente hacen inserciones lógicas de otros objetos. Posteriormente otras reglas que trabajan sobre los objetos originales y lógicos deberán validar dichas sospechas. Es el propio motor que mantiene la validez de las "alertas" según las modificaciones que se hagan en la memoria de trabajo.

En el ejemplo planteado se verifican tres bloques de ejecución claros de disparo de reglas utilizando este recurso:

- Ejecución de reglas que realizan la inserción lógica de otros objetos.

- Activación y ejecución de reglas que realizan operaciones mientras la condición lógica sea válida.
- Activación y ejecución de reglas cuando se vuelve a un estado "normal" de ejecución.