Desarrollo de un Diseñador de reglas de negocio a partir de modelos

Víctor Alberto Iranzo Jiménez

Índice

1. Introducción

1.1. Objetivo del trabajo

El **objetivo** de este trabajo es desarrollar un diseñador de reglas de negocio que sea lo más cercano posible al lenguaje natural. El lenguaje que se utilizará estará restringido: con gramáticas incontextuales se definirán las construcciones que se pueden hacer y estas serán ofrecidas al usuario a través de sugerencias. Las gramáticas se harán empleando la herramienta ANTLR y serán autogeneradas a partir de modelos de entidades.

En la figura 1 se aprecia un prototipo del diseñador: la condición se está construyendo para una entidad de partida específica, el 'Usuario', que ha sido seleccionado con el desplegable superior. El usuario de la aplicación solo puede escribir texto libre cuando se encuentra entre ciertos caracteres, como son las comillas o los corchetes cuadrados, que se emplean para identificar valores numéricos o cadenas de texto a buscar. En el resto de casos, la regla se escribe a partir de la selección que se hace en el listado inferior, que se actualiza conforme avanza la construcción de la regla. Por ejemplo, si lleva escrita la parte de la regla "Si el número de caídas", las sugerencias que le aparecerán abajo serán para realizar comparaciones numéricas, tales como 'es igual a' o 'es mayor que'.

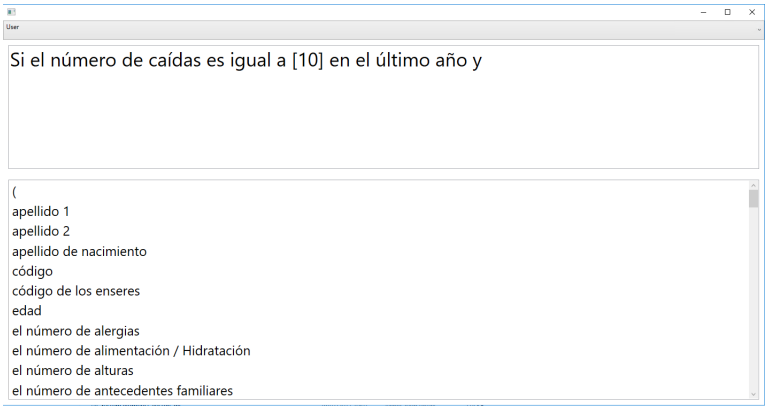

Figura 1: Construcción de una regla partiendo de la entidad Usuario

A continuación, presentaremos brevemente los beneficios del uso de reglas de negocio, junto con las herramientas que se emplean en el desarrollo del caso de estudio. Después, se presentará el desarrollo del mismo siguiendo las etapas del ciclo de vida de desarrollo de software, centrándonos en las partes más relacionadas con la ingeniería del lenguaje natural. Por último, presentaremos las conclusiones del trabajo realizado.

1.2. Reglas de negocio

Las **reglas de negocio** son declaraciones que especifican bajo que condiciones se deben realizar ciertas operaciones en un proceso. Se pueden representar en diferentes niveles de formalidad: desde la redacción escrita en **lenguaje natural**, dentro de un documento de análisis de un sistema, hasta su inclusión en un motor de procesos, que evalúa cambios y ejecuta tareas automáticamente. Algunos de los beneficios de emplear reglas de negocio, con un nivel de automatización avanzado, son los siguientes: [1]

- **Reducción de costes**: los procesos y reglas de negocio pueden ser desarrollados por cualquiera que entienda el lenguaje, gráfico o textual, que se emplea para modelarlos. Las he-

rramientas que permiten modelar procesos de negocio se suelen basar en notaciones gráficas estandarizadas como son **BPMN**, por lo que su curva de aprendizaje no es tan elevada. [2]

- **Mayor control de procesos**: modificar un proceso modelado para adaptarlo a cambios en requisitos es más ágil que modificar la base de código en la que este se traduce. Además, es más fácil detectar errores en el proceso porque pueden representarse gráficamente y se alinea claramente con las estrategias que sigue el negocio.
- **Reusabilidad de componentes**: las condiciones y actividades que se definen en un proceso de negocio pueden reutilizarse en otros.

Siguiendo la definición que hace la herramienta IBM Business Process Manager, que es una conocida plataforma para la gestión de procesos de negocio, una regla de negocio se compone de 2 partes: una condición y una acción. [3] La condición es la situación o estado que debe cumplirse para que ocurra la acción especificada. En este trabajo nos centraremos únicamente en la construcción de **condiciones** en lenguaje natural y quedará excluida la gestión de actividades o acciones, más vinculadas con la gestión orientada a procesos (BPM).

1.3. ANTLR: una herramienta para lenguajes formales

ANTLR (ANother Tool for Language Recognition) es una herramienta para la generación de parsers que permiten la lectura, procesamiento y traducción de texto y ficheros binarios. Puede emplearse para traducir código a otro lenguaje de programación, crear intérpretes de lenguajes específicos de Dominio o crear compiladores. [4]

A partir de una gramática (fichero .g4) que describe formalmente un lenguaje, ANTLR genera un parser, que permite interpretar texto escrito en ese lenguaje. De este texto se construye en ejecución una estructura de datos que puede representarse en forma de árbol (AST - abstract syntax tree). Además, ANTLR genera código para que el árbol construido a partir de la expresión podamos recorrerlo en orden o post-orden, ejecutando cualquier código del cliente al entrar o salir de un nodo. En la figura 2 se observa la estructura que sigue una condición haciendo uso de una de las gramáticas que hemos elaborado. El nodo *simpleCondition* es una regla del parser que representa una condición sobre un campo, el 'código', donde se compara que su valor sea igual a 'A01'. En el árbol, la concatenación de los nodos hoja representa la condición que hemos construido.

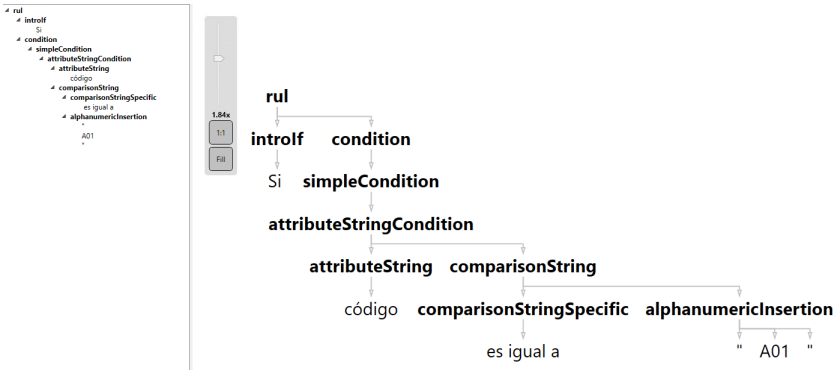

Figura 2: Estructura en árbol de una condición

En nuestro caso de estudio, vamos a hacer uso de esta herramienta para construir un traductor. El traductor transformará la estructura en árbol del AST en una expresión lambda que pueda ser evaluada por nuestro motor de reglas. Los pasos que se siguen para este propósito son los siguientes:

1. Primero, construiremos la gramática que nos permitirá reconocer condiciones. Parte de la estructura de la gramática la detallaremos en **2.3.1 Estructura de la gramática de una entidad**.

2. Una vez construida la gramática, se debe generar el código que se empleará para reconocer una sentencia como válida. Para ello, se hace uso del comando **antlr4 Grammar.g4**. Este código puede ser generado, empleando la opción *-Dlanguage* en diferentes lenguajes de programación: Java, C#, JavaScript o Python entre otros.

3. Para construir nuestro traductor, el código generado por ANTLR debe ser mezclado con código que reconozca nodos como los *simpleCondition* y genere una expresión booleana. Para el recorrido del árbol AST, ANTLR propone dos mecanismos: los *listeneres* y los *visitors*. La diferencia más relevante entre ellos es que para un listener se generan dos métodos para cada tipo de nodo: ambos son invocados automáticamente por la infraestructura de ANTLR (en concreto, el objeto *walker* que debe emplearse), el primero cuando se alcanza el nodo (enter) y el segundo cuando deja de estar activo (exit). En el caso de los visitors, solo se genera un método para cada tipo de nodo y estos métodos **no son invocados automáticamente** cuando se recorre el árbol, sino que debe orquestarse por el código cliente. Por ser más sencillo de implementar, el código de traducción lo escribiremos empleando un listener. [5]

1.4. DSL Tools de Visual Studio

Las DSL Tools de Visual Studio son un conjunto de herramientas dentro del SDK de Modelado que permiten la construcción de lenguajes específicos de Dominio (DSL). En primer lugar, permiten el metamodelado de los conceptos del Dominio y la vinculación de cada uno de ellos con diferentes elementos gráficos para definir cómo se visualizan. Una vez hecho esto, se pueden crear todos los modelos que se quieran empleando el lenguaje gráfico y se puede generar código u otros artefactos a partir de ellos. Además, incluye un motor de validaciones que permite comprobar restricciones sobre los modelos y devolver errores si estos no son válidos. [6]

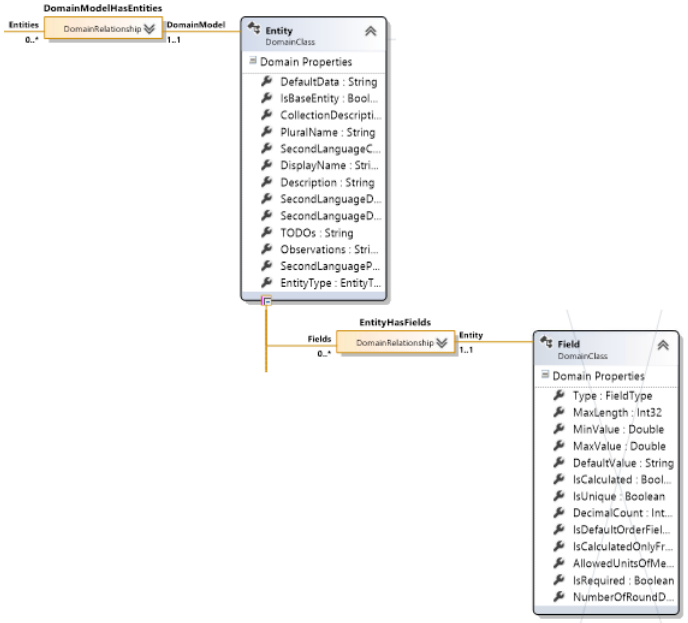

Figura 3: Fragmento del metamodelo donde se muestran algunas propiedades de las metaentidades Entidad y Campo

En nuestro caso de estudio, vamos a hacer uso de una herramienta DSL que hemos desarrollado para representar entidades de Dominio como si un diagrama de clases se tratara. Esta herramienta nos permite representar información de las entidades como su nombre, los campos que la componen y sus tipos, etc. Además, se pueden representar 2 tipos de relaciones entre las entidades: las asociaciones, con su respectiva cardinalidad, y las generalizaciones, que representan la herencia entre entidades. En la figura 3 se muestra un fragmento del metamodelo que se emplea, donde destacamos propiedades como FieldType en Field, que toma como valores tipos primitivos como String o DateTime.

Como hemos dicho, el metamodelo se instancia para producir modelos como el de la figura 4. A partir de este modelo se puede auto-generar código relacionado con la Persistencia o la API de Datos de un servicio. La generación de código se hace mediante plantillas, donde los conceptos que hemos modelado se instancian como objetos. En el contexto de nuestro caso de estudio, emplearemos el modelo para auto-generar una gramática ANTLR de cada una de las entidades. La gramática atenderá a los campos que tiene la entidad, su tipo y las asociaciones entre entidades. Por ejemplo, para la entidad Person se podrán hacer condiciones con el campo 'Name' como es 'Si el nombre es igual a X'. Con su relación con la entidad 'IdentificationDocument' se podrían hacer condiciones como: 'Si tiene algún documento identificativo'.

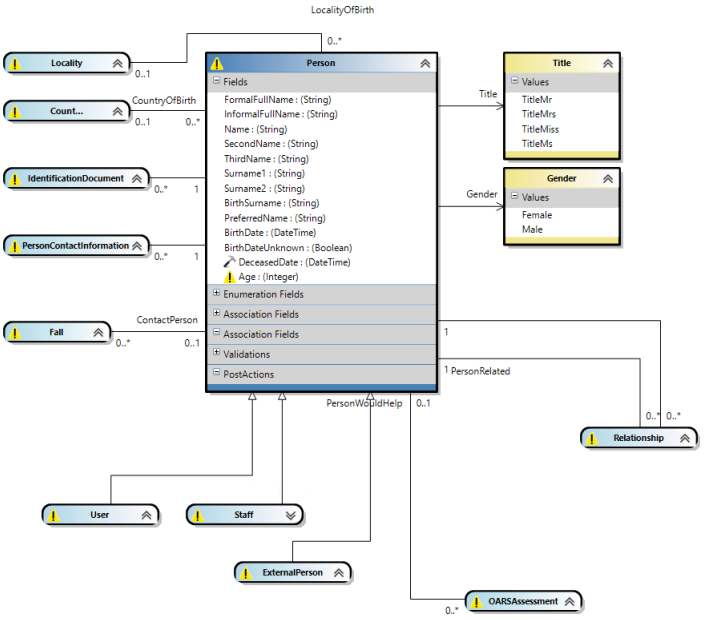

Figura 4: Ejemplo de un modelo de Dominio

2. Desarrollo de un Diseñador de reglas de negocio

2.1. Requisitos del diseñador: hacia el lenguaje natural

El diseñador que se está implementando va a sustituir a uno ya existente y que están empleando los clientes. El motivo principal para reemplazar el diseñador existente es que no utiliza un lenguaje natural que **cualquier usuario** pueda entender. El diseñador está integrado en una aplicación de gestión (ERP) del ámbito socio-sanitario, por lo que muchos de sus usuarios son gente con bajos conocimientos de informática como enfermeras o auxiliares de enfermería.

Una regla que podemos generar con este diseñador es la siguiente: "Si existe un antecedente donde (la fecha del último ingreso en ([0], día/s) es posterior o igual a May 05,2020 y desde ese momento en ([0], día/s) es igual a May 01, 2020) y está en el centro es verdadero". Si la analizamos, vemos que es difícil de entender por el tratamiento de las fechas, el uso de paréntesis innecesarios o el uso de booleanos incorrecto (se podría decir simplemente 'está en el centro').

Otros requisitos interesantes del nuevo diseñador es poder incorporar los artefactos que genera dentro de un diseñador y motor de procesos, como los que hemos descrito en **1.2 Reglas de negocio**.

2.2. Diseño del sistema

En este apartado vamos a definir los diferentes componentes que participan en el proceso de diseño y traducción de una regla, que quedan resumidos en la figura 5.

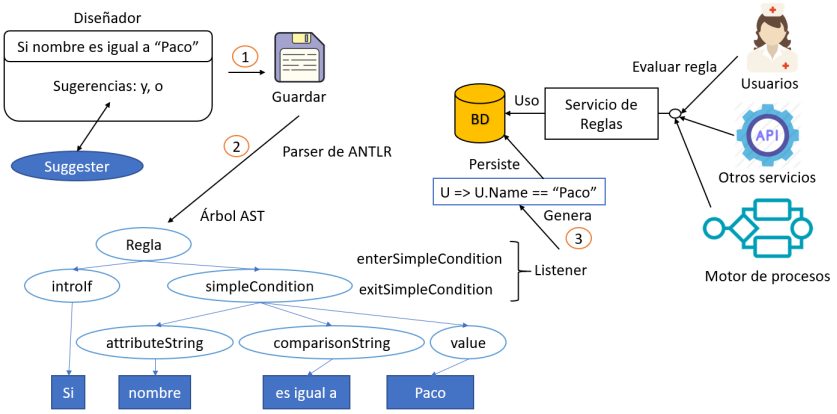

Figura 5: Interacción entre los componentes

Una vez el usuario ha diseñado la regla empleando una interfaz de usuario como la de la figura 1 y le da a Guardar, el texto de la regla es interpretado empleando el parser de ANTLR para obtener el árbol AST de la regla. La gramática sigue una estructura tal que, al emplear un listener de ANTLR, son pocos los nodos sobre los que tendremos que implementar los métodos del listener. Uno de ellos es el de *simpleCondition*, que se transformará en una expresión booleana. Otros nodos importantes a considerar son los *andCondition* y los *orCondition*, que agrupan condiciones simples.

En cualquier caso, el resultado del paseo por el árbol resultará en la generación de una expresión lambda que tiene como punto de partida la entidad de la gramática. Por último, la expresión lambda es persistida en base de datos, que será recuperada por el motor de reglas cuando un usuario, servicio o motor de procesos quiera evaluarla.

2.3. Implementación del diseñador de reglas

En este apartado no vamos a poner el foco en la tecnología empleada. Sí lo vamos a hacer en algunos componentes. Uno de los puntos críticos del sistema es el componente dedicado a nutrir las sugerencias a partir de la gramática conforme se construye una regla. ANTLR **no ofrece directamente esta funcionalidad**, que sin embargo si que hemos podido construir aprovechando que el código de la herramienta es abierto y siguiendo las recomendaciones de posts como el de Federico Tomassetti [7] o el código de Oran Epelbrum [8].

El funcionamiento de este componente al que denominamos **Suggester** es el siguiente. Primero, la porción de la regla escrita es provista al lexer que se genera de la gramática. En nuestro caso, como no permitimos la escritura libre, el resultado será una colección de tokens completos (esto es porque no se proporciona al componente palabras incompletas como 'nombr'). Con los tokens, se recorre en orden la máquina de estados interna que constituye al parser, conocida como ATN. El ATN define, dado un estado, cuales son los posibles tokens que le pueden seguir. Una vez se han consumido en orden todos los tokens que llevamos escritos, se habrá alcanzado un estado que tendrá como transiciones los tokens que devolveremos como sugerencias.

En la práctica, hemos observado que para que funcione correctamente necesitamos eliminar cualquier rastro de **ambigüedad** en la gramática. Esto lo conseguimos mediante la duplicidad de reglas de parsers. Es decir, reglas de parser como son 'comparisonString' y 'comparisonNumeric' que contienen algunos tokens similares como 'es igual a' no pueden agruparse o comenzarían a darse sugerencias incorrectas. Otra forma de evitar estas sugerencias incorrectas podría consistir en el uso de redes neuronales, de tal forma que se entrene un modelo a partir de reglas válidas o de las elecciones que hacen los usuarios en cada paso para ponderar aquellas que no se emplean porque son incorrectas.

2.3.1. Estructura de la gramática de una entidad

Hemos mencionado ya algunas reglas de parser como son *simpleCondition*, *orCondition* o *andCondition*. La primera constituye una condición en si misma sobre un campo de la entidad o una entidad relacionada mientras que las otras dos son una composición de la primera.

Otras reglas de la gramática son importantes, sobre todo las relacionadas con los campos, como es *stringCondition*. Todos los campos de una entidad de un mismo tipo se agrupan en una regla, por ejemplo, *attributeString* en la entidad User tendrá campos como 'Nombre' o 'Primer Apellido'. A continuación, se presenta el tipo de comparación que se puede realizar. Siguiendo el ejemplo, *comparisonString* tendrá tokens como 'empieza por' y no tendrá otros como 'es mayor que', que se emplea en los campos numéricos. Por último, para los valores con los que se compara se emplean caracteres que representan la parte de la regla donde el usuario puede escribir. Estos son las comillas o los corchetes cuadrados, dentro de los cuales se permite introducir cualquier valor mediante una expresión wildcard como [a-zA-Z]*. En definitiva, tenemos una regla como: "stringCondition : attributeString comparisonString anyValue".

3. Valoración personal

Este proyecto es muy **ambicioso** y su éxito depende fundamentalmente de un buen diseño modular. El diseño modular es lo que ha permitido que el proyecto evolucione y se reemplacen diferentes componentes sin cambiar su interfaz con el resto.

El uso de ANTLR inicialmente vino motivado porque se estudió que era una de las herramientas más empleadas en la construcción de lenguajes formales y porque podía generar código para diferentes lenguajes de programación. Sin embargo, la parte más costosa y desafiante ha sido la construcción del Suggester y es una funcionalidad que podría ofrecer la herramienta de forma nativa. Por otro lado, otro aspecto negativo que no nos ha gustado de ANTLR es el gran tamaño del parser que construye, en lo referido a líneas de código. Esto por supuesto es debido a que algunas de nuestras gramáticas también son muy grandes, sobre todo aquellas donde se hace un uso intensivo de la herencia. Por ejemplo, en asociaciones con entidades base se puede hacer condiciones con cualquiera de las sub-entidades, lo que aumenta considerablemente el tamaño de la gramática.

Estos aspectos negativos han hecho que planteemos, de nuevo, cambiar algunos componentes del sistema. Por ejemplo, la gramática, que consideramos un objeto muy estático, podría reemplazarse por un componente más dinámico, de tal forma que se generen las sugerencias conforme se recorre el modelo de Dominio a partir de una entidad de partida. Esto será necesario finalmente debido a que queremos poder generar condiciones partiendo de una entidad y navegando hacia otras con un grado de profundidad elevado.

4. Conclusiones y trabajo futuro

El uso de generación automática de código hace que podamos generar gramáticas de forma **homogénea** para cualquier entidad del modelo. Además, su evolución será más ágil: solo será necesario cambiar las plantillas que originan la gramática para introducir nuevos requisitos o corregir errores, y ese cambio se propagará inmediatamente a todas las gramáticas existentes.

Sin embargo, el prototipo generado sigue todavía lejos del lenguaje natural. Esto se debe a que se emplea el mismo esqueleto para auto-generar las gramáticas de todas las entidades, cuando algunos verbos o construcciones no encajan igual de bien dentro de lo que se considera correcto en el Dominio sanitario.

Por último, ANTLR se ha visto un mecanismo útil en la generación de la gramática empleada para crear reglas de negocio. Si bien puede resultar **rígido** para interpretar texto libre, en un diseñador asistido y guiado como el presentado resulta acertado. No obstante, su uso dentro del proyecto está siendo cuestionado y se están evaluando alternativas para reemplazarlo.

Referencias

[1] Wallace Oliveira. ¿Qué son las reglas de negocio y cuáles son las ventajas de aplicarlas en una empresa?, 2019. URL https://www.heflo.com/es/blog/automatizacion-procesos/que-son-reglas-negocio/.

[2] BPMN Specification - Business Process Model and Notation. URL http://www.bpmn.org/.

[3] Reglas de negocio. URL https://www.ibm.com/support/knowledgecenter/es/SSFTN5_8.5.6/com.ibm.wbpm.wid.bpel.doc/busrules/topics/cundbus.html.

[4] Terence Parr. *The Definitive ANTLR 4 Reference*. 2013.

[5] Antlr4 - Visitor vs Listener Pattern - Saumitra's blog. URL https://saumitra.me/blog/antlr4-visitor-vs-listener-pattern/.

[6] Herramientas de los lenguajes específicos de dominio - Visual Studio — Microsoft Docs. URL https://docs.microsoft.com/es-es/visualstudio/modeling/overview-of-domain-specific-language-tools?view=vs-2019.

[7] Federico Tomassetti. Building autocompletion for an editor based on ANTLR. URL https://tomassetti.me/autocompletion-editor-antlr/.

[8] Oran Epelbrum. Java auto-suggest engine for ANTLR4 grammars. URL https://github.com/oranoran/antlr4-autosuggest.