UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO

Entorno de Pruebas de Generadores de Código Automático

José Carretero Arias

Tutor: Jesús Sánchez Cuadrado

Mayo-Junio 2013

PALABRAS CLAVE

A continuación se muestra el listado de palabras clave que se consideran más relevantes para este documento:

- Desarrollo de Software Dirigido por Modelos
- Generación de código automática
- Lenguaje específico de dominio
- Pruebas de generadores de código
- Generador de documentación

KEYWORDS

Below is a list of keywords which are considered most relevant to this document:

- Model-driven engineering
- Automatic Code Generation
- Domain-Specific Language
- Code generator testing
- Documentation generator

RESUMEN

El presente documento constituye la memoria del proyecto realizado como trabajo de fin de grado. El proyecto tiene como objetivo el desarrollo de un entorno de pruebas de generadores de código automático, así como también la implementación de un generador de documentación que permita de manera automática documentar generadores de código en forma gráfica.

Este proyecto surge en el contexto de unas prácticas realizadas en la empresa CAF Signalling, en las que se desarrollaron dos generadores de código para máquinas de estado, utilizando el paradigma conocido como de Software Dirigido por Modelos. Durante su realización se observó la necesidad de probar los generadores de código realizados y posteriormente documentarlos.

Para llevar a cabo las pruebas sobre los generadores se optó por la creación de un entorno de validación que fuera genérico para realizar pruebas sobre cualquier tipo de generador. A lo largo del documento se detallan las técnicas y soluciones adoptadas para resolver la necesidad observada.

En primer lugar, y tras introducir el contexto en el que se desarrolla el proyecto, la motivación del mismo y el estado del arte, se explica el diseño de un lenguaje específico de dominio ideado para dar solución al problema y se detallan los tres componentes básicos que lo componen: el metamodelo, una sintaxis textual que permita escribir especificaciones y, finalmente, un intérprete para realizar las pruebas especificadas. Todo ello se desarrolla con la ayuda de las tecnologías *Xtext*, que como se verá es un *framework* especializado en la creación de lenguajes específicos de dominio, y EMF, especializado en metamodelado, y gracias al cual se hace posible la creación de herramientas genéricas como el entorno de validación.

A continuación, se expone el diseño y desarrollo del generador de documentación. Se han creado dos algoritmos para recorrer un generador de código escrito con *Acceleo* y extraer una representación gráfica de los mismos para la herramienta *Graphviz*.

Finalmente se muestran las pruebas realizadas y los resultados obtenidos tras aplicar ambos prototipos de herramienta en un proyecto real.

ABSTRACT

This document reports on the work done as Final Degree Project. The project aims at developing a test environment for automatic code generators as well as implementing a documentation generator which automatically document code generators in a graphical form.

The project comes within the context of an internship carried out in the company CAF Signalling, in which two code generators for state machines were developed using the Model-Driven Software Engineering paradigm. During their implementation it was observed the need to test the code generators made and subsequently documenting.

The approach for testing the generators has been the creation of a validation environment that is generic, in the sense that is able to specify tests for any type of generator. Throughout the document the techniques and solutions adopted to realized the approach are detailed.

First, but after introducing the context in which the project is developed, its motivation and the state of the art, the design of a domain specific language designed to solve the problem is explained. Also the three basic components that comprise it are detailed: the metamodel, a textual syntax that allows writing specifications and finally, an interpreter to carry out the specified tests. All of this takes place with the help of Xtext technology, which is a specialized framework in creating domain specific languages, and EMF, specialized in metamodelling and by which is posible to create generic tolos such as this validation environment.

Then, the document describes the design and development of the documentation generator. To implement it two algorithms for traversing an existing code generator and draw a graphical representation of itusing Graphviz has been developed.

Finally the tests performed and the results obtained after applying both prototyping tool in a real project are shown.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

En la actualidad existen labores, llevadas a cabo por empresas, que conllevan realizar tareas de programación repetitivas y propensas a que el desarrollador cometa errores. El código resultante de dichas tareas es susceptible en muchas ocasiones de ser generado parcial o totalmente de manera automática.

En este contexto surgió, a principios de la década pasada, un nuevo paradigma conocido *Desarrollo de software dirigido por modelos* (DSDM) [1], con dos propósitos fundamentales: producir el salto definitivo hacia la industrialización del software y proporcionar mejoras importantes en la productividad y calidad del mismo.

El pilar fundamental en el que se basa el DSDM es la generación de código de manera automatizada a partir de modelos expresados mediante lenguajes de modelado (por ejemplo UML [2]) o con *Lenguajes Específicos de Dominio* (DSL) [3]. De esta forma, se pueden desarrollar generadores de código que resuelvan problemas concretos facilitando las tareas anteriormente mencionadas, ahorrando así tiempo y coste a las empresas.

Este proyecto ha sido realizado en el contexto de unas prácticas en empresa, en particular en un proyecto desarrollado para la empresa CAF Signalling. En el mismo llevé a cabo el desarrollo de un generador de código automático para la generación de código Java. A partir de modelos de máquinas de estado se genera código Java dirigido a la gestión y control de sistemas ferroviarios.

Uno de los inconvenientes que se encontró al realizar el generador fue cómo determinar si lo que se había generado era correcto. Para solventar este inconveniente se hace necesaria una herramienta que valide y garantice que el código generado se corresponde con el código esperado. Otro problema que se detectó fue la necesidad de documentar los generadores de código que se habían realizado, ya que era necesaria algún tipo de notación para ello e idear algún mecanismo para extraer la documentación de los generadores ya creados.

En los siguientes apartados se lleva a cabo tanto la descripción del generador de código desarrollado, como de un entorno genérico de validación de generadores de código y de documentación, el cual constituye el objeto central del TFG.

En primer lugar se describirá el contexto en que se ha realizado el proyecto, realizando una introducción al DSDM como nuevo y reciente paradigma objeto de investigación y desarrollo (Sección 2). A continuación, en la Sección 3, se presentará un resumen del estado del arte en pruebas de generadores de código así como como también las principales tecnologías implicadas y utilizadas en el proyecto.

En la Sección 4 se detallará el diseño y desarrollo del entorno de validación, especificando los distintos módulos desarrollados junto con su funcionalidad. También se describirá la aproximación ideada para documentar generadores de código.

Finalmente se detallarán las pruebas realizadas y los resultados obtenidos tras su aplicación en un proyecto real (Sección 5) y se mostrarán las conclusiones extraídas y posibles mejoras sobre el proyecto (Sección 6).

2. CONTEXTO Y MOTIVACIÓN

En los siguientes apartados se introduce el contexto en el que se desarrolla este proyecto, en concreto se estudia el paradigma de la generación de código automática dirigida por modelos junto con las tecnologías utilizadas para el desarrollo del generador de código y del entorno de validación de generadores de código.

2.1. DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS (DSDM)

Tal y como se indicó en la introducción, los principales objetivos por los que surgió el *Desarrollo de Software Dirigido por Modelos* (DSDM) fueron producir el salto definitivo hacia la industrialización del software y proporcionar mejoras importantes en la productividad y calidad del mismo. Con ello, se pretende que se abandonen "las prácticas artesanales" en el desarrollo de software y se produzca la transición hacia una producción industrial, caracterizada por la automatización de los procesos de desarrollo, la utilización de herramientas que permitan automatizar tareas repetitivas y crear posibles líneas de producto.

El DSDM implica elevar el nivel de abstracción en el desarrollo de software dándole una mayor importancia al modelado conceptual y al papel de los modelos en el desarrollo del software. Además, aporta numerosos beneficios al desarrollo como por ejemplo:

- Mejora de la productividad gracias a la automatización del proceso que permite la generación de gran parte de los artefactos de la aplicación.
- Aumento de la calidad debido a la generación de código de forma automática que garantiza que el código cumpla un mismo estándar y se reduzcan los errores.
- Mejor comprensión del sistema a desarrollar. Con ayuda de visualizaciones de los modelos en forma de grafo o con lenguajes textuales más compactos y expresivos que los lenguajes de propósito general, se obtiene una visión global del sistema en un lenguaje claro e intuitivo.
- Facilita evolución y mantenimiento puesto que se describen procesos estándar y automatizados que siguen una reglas más fáciles de mantener.

- Facilita reuso/reimplementación en otras tecnologías ya que al basarse en un mayor nivel de abstracción, se pueden adaptar con gran facilidad.

Como ejemplo de aplicación del DSDM, en el proyecto que ha servido de marco para la realización de este trabajo se implementó un generador de código para la generación de código Java. A partir de máquinas de estado modeladas en UML, las cuales constituyen la entrada al generador, se genera el código Java que implementa la lógica de funcionamiento de dichas máquinas de estado.

De igual forma, como entrada al entorno de validación de generadores de código, se utilizan los mismos modelos con los que se ha generado el código y se realizan las validaciones pertinentes sobre el código generado. Como se verá más adelante, dichas validaciones son especificadas por el programador en un formato definido en un nuevo DSL creado para especificar qué se desea validar.

En los siguientes apartados se describe el papel que juegan los modelos en el paradigma de la generación de código y de qué manera se relacionan con el entorno de validación de generadores de código.

2.2. METAMODELOS Y MODELOS

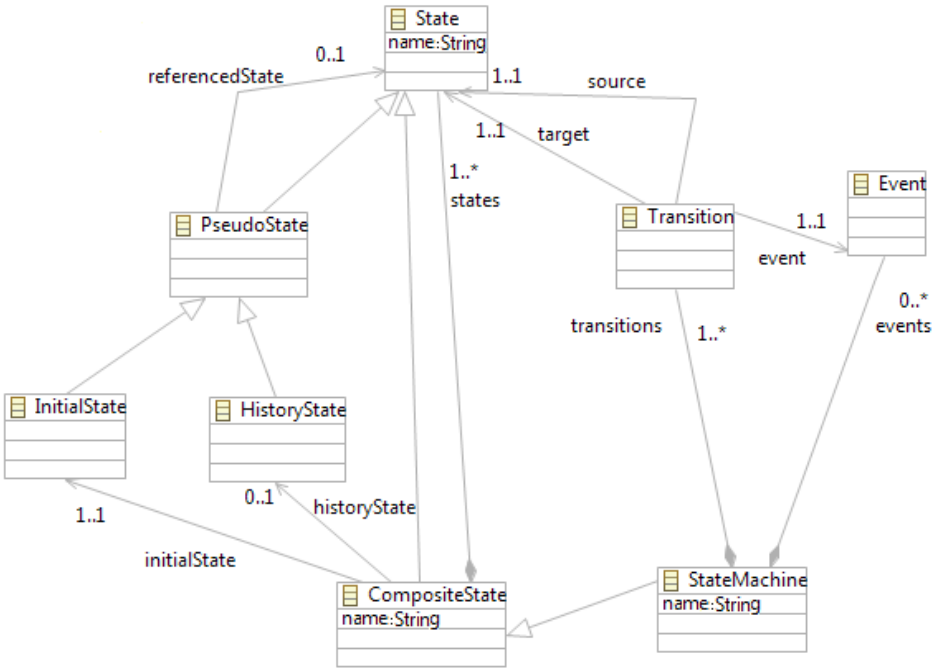
Un modelo es una descripción de un sistema bajo estudio utilizando un lenguaje. Las reglas de construcción de este lenguaje se representan mediante un metamodelo, que describe la sintaxis abstracta del lenguaje.

Los modelos son instancias válidas de metamodelos. Válidas a nivel estructural, lo que quiere decir que los objetos que lo componen deben ser de clases definidas por el metamodelo; y válidas a nivel restrictivo cumpliendo todas las restricciones que imponga el metamodelo.

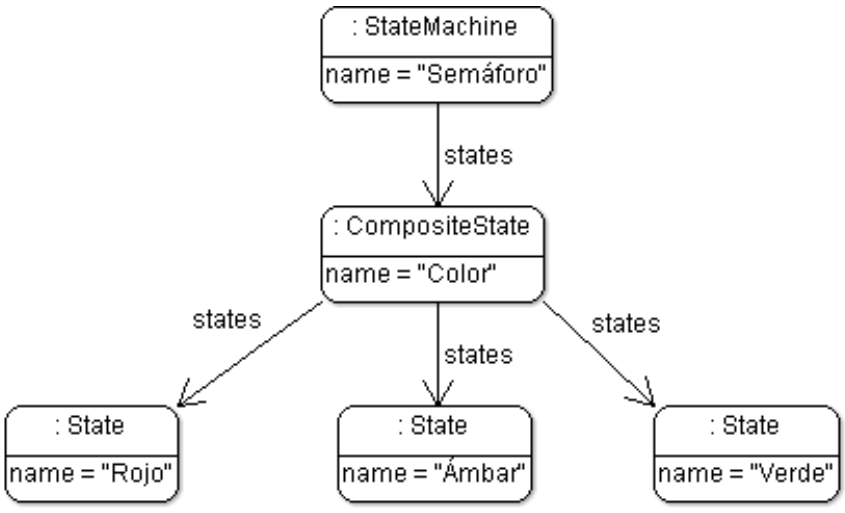
En el contexto del proyecto realizado con la empresa CAF Signalling se utilizaba el metamodelo de las máquinas de estado de UML. A partir de máquinas de estado modeladas según el metamodelo, se genera código Java destinado a la gestión y control de sistemas ferroviarios.

Además, para construir el DSL utilizado en el entorno de validación en el que se especificarán las validaciones a realizar sobre el generador de código, se ha creado un metamodelo específico representar las construcciones de ese lenguaje.

Para entender mejor estos dos conceptos, se muestra a continuación un ejemplo de metamodelo y una instancia (modelo) de él, en concreto una versión simplificada del metamodelo de máquinas de estado de UML (Figura 1a), y un ejemplo de modelo que es una instancia del meta-modelo (Figura 1b).



(a) Metamodelo de Máquinas de Estado



(b) Modelo (instancia válida del metamodelo)

Figura 1. Metamodelo y Modelo

2.3. SINTAXIS CONCRETA

En la **figura 1**, se ha visto la representación de un metamodelo y un modelo tal y como se representa en memoria. No obstante, el usuario normalmente prefiere trabajar con otro tipo de representación del modelo, es decir, utilizar una notación que le permita especificar modelos de manera sencilla.

Las dos formas más comunes de representación de modelos con las que suele trabajar el usuario son la gráfica y la textual.

A continuación se muestra el mismo ejemplo de modelo representado en la **figura 1,** representado en una posible sintaxis textual:

```
StateMachine: Semáforo {
        CompositeState: Color {
                State Rojo;
                State Ámbar;
                State Verde;
        }
}
```

El mismo ejemplo en formato gráfico quedaría de la siguiente forma:

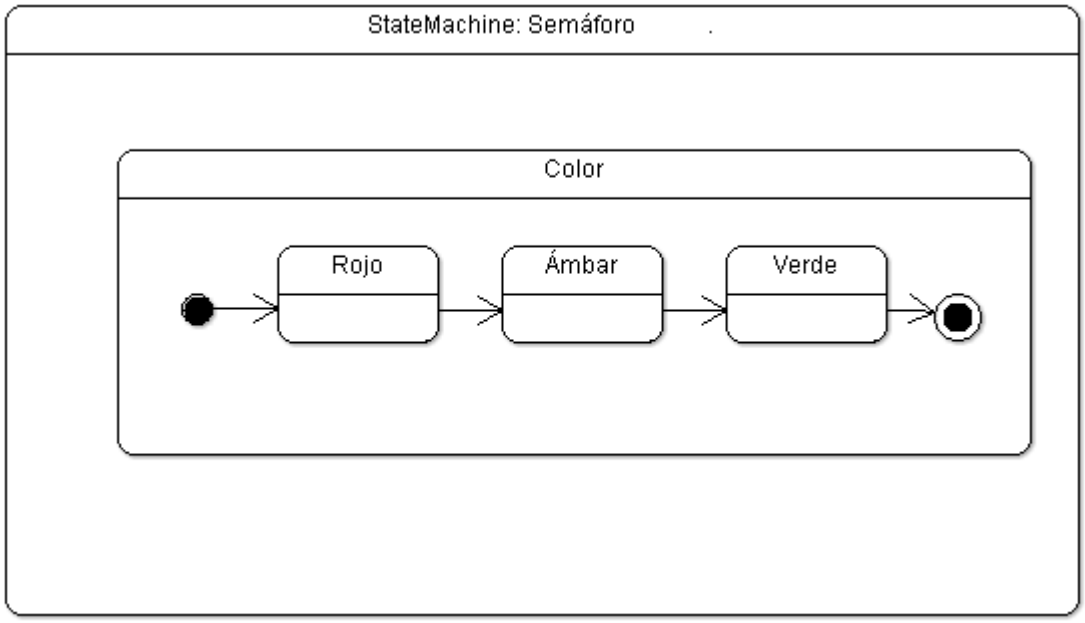


Figura 2. Representación gráfica del modelo

2.4. TRANSFORMACIÓN DE MODELOS

Con el paradigma del DSDM surge la idea de desarrollar software capaz de realizar la lectura de los modelos y ejecutar las acciones correspondientes en función de la tarea que se deba realizar: los generadores de código. Estos se basan en transformaciones de los modelos.

En el DSDM existen dos tipos de transformaciones fundamentales, transformaciones modelos y transformaciones modelo-a-texto (también conocidas como generadores de código). Las primeras permiten modificar un modelo de entrada o convertirlo a un modelo conforme a un metamodelo distinto, y son muy utilizadas para refactorizar u optimizar modelos, realizar conversiones entre lenguajes o como paso previo a la generación de código.

Una transformación modelo-a-texto o generador de código es un tipo particular de transformación que toma como entrada uno o más modelos y genera uno o más ficheros de texto como salida. Normalmente estos ficheros contienen código en algún lenguaje de programación o son ficheros de configuración como XML.

Por lo general, un generador de código se concibe para dar solución a un problema concreto, no obstante, existen tecnologías como *Acceleo*, que facilitan el desarrollo de un generador de código ofreciendo construcciones específicamente diseñadas para construirlos. Para ver cómo funciona una transformación, se explica muy brevemente el uso de la herramienta *Acceleo.*

Acceleo se organiza en módulos y plantillas que realizan llamadas entre sí. Una plantilla puede contener texto estático escrito directamente, que se generará de igual forma en todas las generaciones, y texto dinámico que variará en función del modelo de entrada en cada generación. Para ver esto, se expone un pequeño ejemplo de plantilla en el que por cada estado que se encuentre se debe generar el prototipo de una función:

```
[template public generaPrototipoFuncionEstado (estado : State, clase : Class)]
Boolean_te en_[clase.name/]_funEstado[estado.name/] ();
[/template]
```

Cada vez que se llame a esta plantilla, se generará una línea de código que se corresponde con el prototipo de función para ese estado. El código que se genera de

manera dinámica es el que aparece entre "**[/]**", que se sustituye por el valor que tenga el modelo para ese estado.

2.4. VALIDACIÓN DEL CÓDIGO GENERADO AUTOMÁTICAMENTE

Durante el desarrollo del generador de código para código Java a partir de máquinas de estado, se observó una problemática. Conforme aumenta la cantidad de código a generar o se hace más complejo el sistema a automatizar, se es más susceptible de cometer errores. Esta problemática dio lugar a la concepción de un entorno que se encargara de validar el código generado verificando algunos aspectos y garantizando así que la generación es correcta.

La idea principal es desarrollar un prototipo de entorno de validación que muestre la potencialidad de una herramienta con características muy útiles en el paradigma de la generación de código automática. Por ejemplo, en el proyecto de CAF Signalling, en el que se desarrolló el generador de código para código Java, era importante mantener la estructura de directorios en los que se organizaba el código original. Tras desarrollar el generador y realizar generaciones de código sobre las máquinas de estado, se observó el problema de que se cometían errores que hacían que la jerarquía de directorios del código generado no era la esperada y no permitía el funcionamiento del sistema. En la **figura 3** se puede observar la estructura exigida por la empresa:

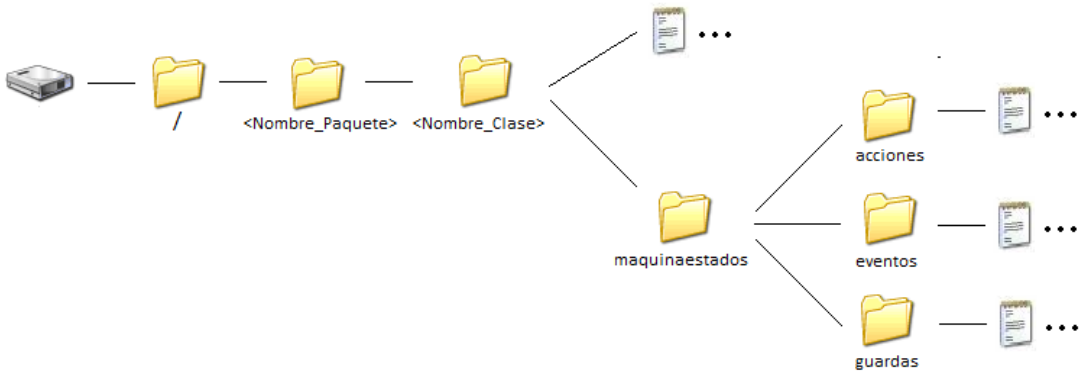


Figura 3. Jerarquía de directorios

Para garantizar que el código generado se organiza como se espera, el validador realiza comprobaciones como pueden ser: validar que las rutas de los paquetes de todos los ficheros son correctas, comprobando a la vez que existan los directorios necesarios, o comprobar que los nombres de fichero se corresponden con los nombres de las clases que contienen. Otros errores característicos eran errores en los nombres de las clases o métodos Java, elementos para los que no se generaba texto (o al revés, que no debía generarse texto pero se generaba), etc.

En el apartado de diseño y desarrollo se explican de manera muy detallada las características con las que se ha dotado al entorno para mostrar su funcionalidad.

En los siguientes apartados se estudia el estado en el que se encuentra el área dedicada a la validación de código generado por generadores de código y la tecnología y herramientas utilizadas para el desarrollo de este proyecto. Además se expone de manera detallada el diseño y desarrollo del entorno de validación creado, así como también las pruebas y resultados obtenidos tras ser aplicado en un proyecto real.

3. ESTADO DEL ARTE

3.1. PRUEBAS Y DOCUMENTACIÓN DE GENERADORES DE CÓDIGO

La transformación de modelos es una de las técnicas claves en el DSDM, puesto que es la manera de manipular modelos automáticamente. La cuestión de cómo probar que las transformaciones resultan de cierta importancia para asegurar que las manipulaciones realizadas son correctas. Muchos de los esfuerzos en este campo están destinados a probar transformaciones modelo-modelo [4], mientras que las pruebas para los generadores de código se consideran como algo relativamente directo.

La razón principal para esto es que en la actualidad existe cierto consenso acerca de las técnicas que deben utilizarse para generar código, existiendo incluso un lenguaje estándar para crear generador de código, MOF-To-Text [5].Sin embargo, como se ha puesto de manifiesto en la realización del proyecto para la empresa CAF Signalling, es importante tener la posibilidad de probar automáticamente que el código generado satisface ciertos requisitos y convenciones.

El principal problema a la hora de probar generadores de código es que la salida es texto y es complicado realizar aserciones sobre el mismo. Por esta razón, la principal técnica para realizar pruebas es la validación cruzada [6]. Esta técnica consiste en crear un generador y disponer de un simulador (o crear dos generadores) para el mismo problema, de manera que dado un modelo se observan los efectos del código generado y del simulador y se comparan los resultados. Si difieren, se debe comprobar si hay errores en el generador. Esta técnica habitualmente se complementa con la generación automática de modelos de pruebas [7]. Otras técnicas que se usan habitualmente en la industria son la revisión de código [8] o los análisis estáticos del código generado [9]. La aproximación seguida en este proyecto es diferente, y trata de aplicar la idea de pruebas unitarias a los generadores de código, mediante la descripción de aserciones o condiciones que debe cumplir el texto generado a través de un DSL.

En cuanto a entornos de pruebas en DSDM, EUnit es un *framework* genérico para probar operaciones de gestión de modelos [10], construido sobre ANT y que por tanto no está integrado con un IDE. Incluye transformaciones modelo-a-modelo, pero no considera los generadores de código. En [11] se presenta un *framework* para probar transformaciones de modelos, pero tampoco tiene en cuenta los generadores de código.

Finalmente, con respecto a aproximaciones para documentar o diseñar transformaciones de modelos existen algunas propuestas como *transML* [12], pero ninguna específicamente orientada a generadores de código. Además, el prototipo creado en este proyecto es capaz de extraer documentación a partir de un generador existente escrito en *Acceleo*.

3.2. HERRAMIENTAS DE DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS

A continuación se exponen, sin entrar en mucho detalle, las herramientas y tecnologías utilizadas durante el desarrollo del proyecto:

2.4.1. ACCELEO

Como tecnología principal para el desarrollo del generador de código se ha utilizado *Acceleo* [13]. Esta herramienta es un lenguaje y un entorno de programación para Eclipse orientados a la transformación de modelos en código.

Acceleo permite tanto generar código estático escrito directamente por el programador, como generarlo de manera dinámica extrayendo la información deseada de modelos previamente cargados en memoria.

Las características fundamentales por las que se eligió esta tecnología frente a otras similares han sido su rápida y fácil integración en forma de *plug-in* para Eclipse y porque un programa *Acceleo* es también un modelo, lo cual permite trabajar de forma más cómoda dentro del DSDM (como se mostrará al presentar el generador de documentación para generadores de código). Además, *Acceleo* es una implementación de estándar MOF-To-Text del OMG [14].

2.4.2. XTEXT

Xtext [15] es un *framework* especializado en el desarrollo de lenguajes específicos de dominio. Como ya se ha explicado anteriormente, éstos permiten especificar un lenguaje dedicado a resolver un problema en particular y proveer una técnica relativamente más sencilla para solucionar dicho problema.

En el contexto de este proyecto, se ha creado un DSL para realizar las especificaciones, por parte del programador, de las validaciones que se desean realizar sobre el generador de código, de tal forma que facilite dicha tarea. Para ello se ha generado una gramática sencilla e intuitiva, que se detallará más adelante, mediante la ayuda del *framework* de *Xtext*.

Xtext permite ligar una gramática a un metamodelo de tal forma que al ejecutarse el parser se instancia el modelo que es la representación en memoria del DSL. Además, genera un entorno en Eclipse para el DSL, que puede personalizarse, y que incluye resaltado de sintaxis, autocompletado, gestión de errores, etc.

2.4.3. ECLIPSE

El proyecto se ha desarrollado en su mayoría en el entorno de desarrollo integrado (IDE) de Eclipse [16] ya que éste ofrece soporte para Java y gran extensibilidad gracias a su arquitectura en forma de *plug-in*.

Tanto el generador de código Java como el intérprete creado para realizar la lectura de las especificaciones de validación han sido desarrollados en código Java. El generador de código se ha encapsulado en forma de *plug-in.* El intérprete, en cambio, como aún se encuentra en fase de prototipo no se ha encapsulado en forma de *plug-in* pero está pensado para serlo.

2.4.3. EMF

EMF (Eclipse Modelling Framework) constituye el estándar de facto para metamodelado [17]. Es la versión más utilizada actualmente de la arquitectura estándar MOF. Ésta contiene una API reflectiva que permite tratar con modelos de manera genérica sin conocer qué tipo de metamodelo se está utilizando y, por tanto, permite también la construcción de herramientas genéricas.

Gracias a su fácil y cómoda integración en forma de *plug-in* para Eclipse, ha servido de gran ayuda para la creación del metamodelo Validador, utilizado para realizar especificaciones de validación en este proyecto.

2.4.4. GRAPHVIZ

Graphviz es un software de código abierto para creación y visualización de grafos. La visualización en forma de grafo permite representar información estructural de manera organizada y más legible.

Como complemento a la validación del generador de código se ha desarrollado un pequeño prototipo de programa que genera la documentación sobre el generador. En concreto, se generan los diagramas estructurales de la jerarquía de módulos que componen un generador de código y de la relación de llamadas entre las distintas reglas de dichos módulos.

4. DISEÑO Y DESARROLLO

En esta sección se exponen los pasos seguidos e hitos marcados durante el proyecto para su realización, así como también la metodología adoptada. También se explican en detalle la problemática observada durante el desarrollo de generadores de código y la solución adoptada para resolverla.

Además se explicará una aplicación adicional creada como prototipo de generación automatizada de algunas partes de la documentación de generadores de código, muy útil en contextos reales donde los generadores de código pueden llegar a ser bastante complejos.

4.1. METODOLOGÍA Y PLANIFICACIÓN

El proyecto ha sido desarrollado durante los meses de Enero a Mayo de 2013. Para alcanzar la meta establecida con mayor garantía de éxito se acordó dividir el proyecto en 4 fases durante la primera reunión con el tutor, Jesús Sánchez. También se acordó que al inicio de cada una de las fases se realizaría una reunión en la que se detallaría el proceso a seguir para la resolución de cada fase y otra reunión para comprobar si los objetivos se habían alcanzado al concluir cada una de las mismas.

A continuación se exponen brevemente las fases establecidas, junto con sus objetivos y sus respectivas fechas de inicio y fin:

4.1.1. FASE 1: ALCANCE DE LAS CARACTERÍSTICAS DE VALIDACIÓN

Fecha inicio: 14/01/2013, Fecha fin: 27/01/2013

En esta primera fase del proyecto se fijó como objetivo pensar y acotar las características que el validador permitiría realizar y mostrar en su versión como prototipo. Éstas se exponen detalladamente más adelante, pero debían ser genéricas para poder aplicarse a distintos generadores y abarcar un abanico de diversidad en su funcionalidad.

4.1.2. FASE2: DISEÑO DEL METAMODELO Y CREACIÓN DE LA GRAMÁTICA

Fecha inicio: 28/01/2013, Fecha fin: 10/02/2013

La segunda etapa establecida tenía como objetivo diseñar el metamodelo del lenguaje de validación, que describiera las construcciones y la semántica del lenguaje.

En esta etapa además del diseño del metamodelo se debería crear la sintaxis concreta textual del DSL que permitiera a los desarrolladores de manera sencilla especificar las validaciones que se desean realizar sobre el código generado.

4.1.3. FASE 3: IMPLEMENTACIÓN DEL INTÉRPRETE

Fecha inicio: 11/02/2013, Fecha fin: 21/04/2013

El tercer hito establecido tenía como propósito la implementación de un intérprete que tomara las especificaciones realizadas por el desarrollador e hiciera efectivas sus correspondientes acciones, dando como salida el resultado de la validación.

En esta tercera fase concluiría el desarrollo del entorno de validación.

4.1.4. FASE 4: IMPLEMENTACIÓN DEL PROTOTIPO DE DOCUMENTADOR

Fecha inicio: 22/04/2013, Fecha fin: 05/05/2013

En la cuarta fase de proyecto se decidió crear un nuevo prototipo de aplicación que se encargara de generar documentación sobre cómo se estructura el generador de código y la relación existente entre llamadas a los distintos módulos que componen el generador.

4.2. ENTORNO DE VALIDACIÓN DE GENERADORES DE CÓDIGO

4.2.1 PROBLÉMATICA Y SOLUCIÓN ADOPTADA

Como ya se ha ido adelantando en apartados anteriores, el desarrollo de generadores de código es susceptible también de contener errores. A medida que la complejidad del sistema aumenta la cantidad de código a generar es mayor o cuando

aparecen nuevos requisitos que deben ser integrados en un generador existente, el desarrollador tiende a introducir errores en el código de salida del generador.

Para dar solución a este problema y reducir la posibilidad de que el código generado contenga errores, se concibió la idea de crear un entorno de validación que fuera genérico, de tal forma que sirva para realizar validaciones sobre cualquier generador de código que se desarrolle para solventar problemas concretos. De este modo se pretende crear una herramienta universal para cualquier generador de código, independientemente del problema que trate, en el que sea el desarrollador el que decida qué cosas desea validar en función de sus necesidades y garantice así que el código generado es el que se espera.

Para dar una visión más clara del sistema, se muestra a continuación un esquema que refleja la relación entre el generador de código y el entorno de validación:

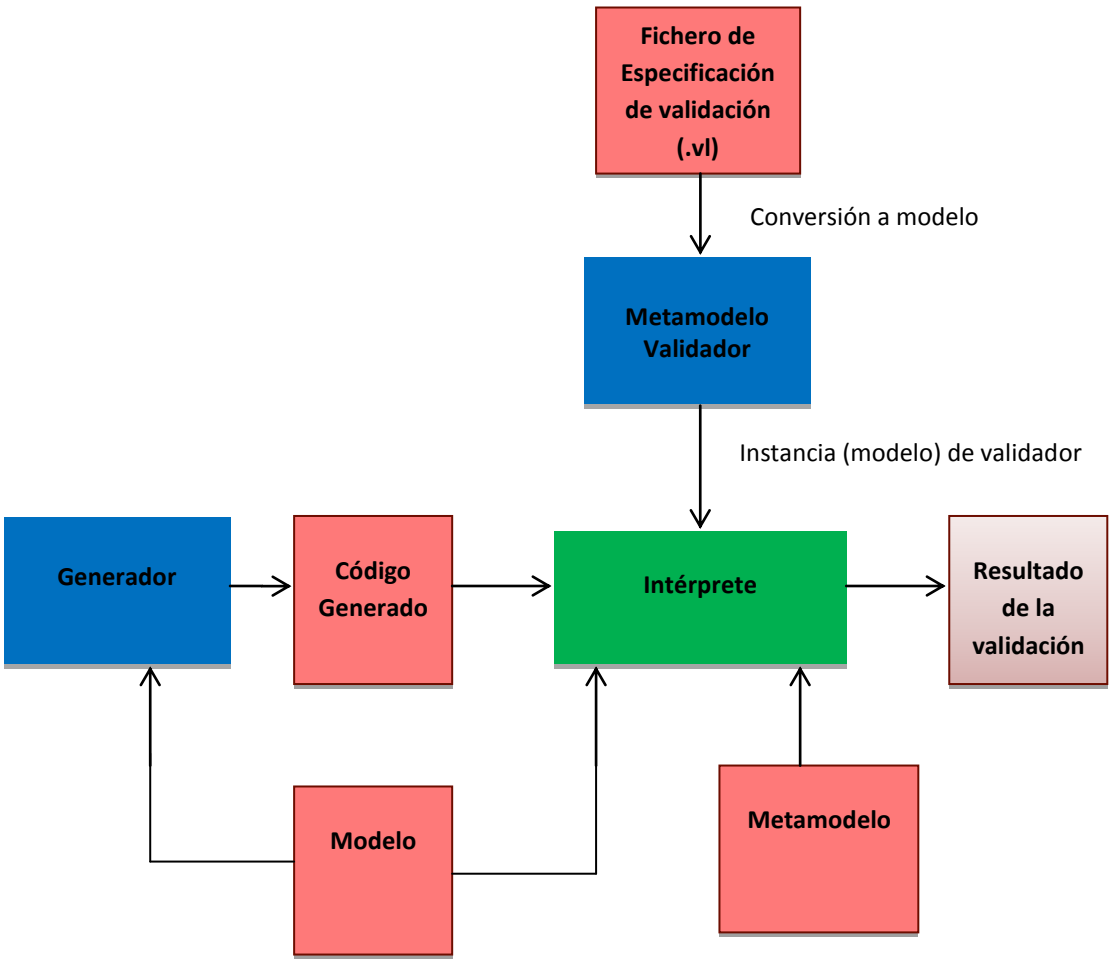


Figura 4. Interacción Generador-Validador

Las acciones a validar descritas por el desarrollador en el fichero de especificación, se procesan para convertirlas en una instancia válida, o modelo, del metamodelo creado Validador, el cuál se explica más adelante. A partir del modelo que contiene las especificaciones a validar, el código generado por el generador y el modelo sobre el que se ha generado el código (junto con su metamodelo correspondiente), el intérprete se encarga de realizar las comprobaciones solicitadas y de generar un informe de resultados.

En los siguientes apartados se describen en profundidad qué características se han implementado en la versión de prototipo, el metamodelo creado para poder instanciar modelos de especificación de validación y la gramática que permite especificar dichas características. También se detalla la implementación interna del intérprete que realiza las comprobaciones pertinentes.

4.2.2 FASE 1: CARACTERÍSTICAS DE VALIDACIÓN

La primera fase del proyecto tenía como objetivo establecer las características que el entorno de validación permitiría validar.

El entorno de validación está pensado para ser genérico y poder realizar validaciones sobre cualquier tipo de generador. Como ejemplo de la necesidad de que el entorno de sea genérico (es decir, no ligado a un tipo concreto de meta-modelo), en el proyecto en el que trabajaba se desarrollaron dos generadores de código para máquinas de estado: el primero un generador de código para código Java y el segundo para código C.

Durante el desarrollo de ambos generadores se observaron errores muy comunes en el código generado que daban lugar a una necesidad de ser validados. Además para tener garantías de que la salida del generador se correspondía con lo esperado, se hacían muy útiles realizar validaciones de ciertas características como se verá a continuación.

Para poder mostrar un prototipo de herramienta universal de validación de generadores de código se han implementado las siguientes características, las cuales serán detalladas cuando se explique el intérprete:

- Validar que el número de ficheros que se generan para un determinado tipo de objeto, se corresponde con el número de ficheros real necesarios de dicho objeto.

- Chequear que la ruta del paquete de un fichero Java es correcta, es decir, se corresponde con la ruta física en la que se encuentra.

- Comprobar que las clases de los ficheros Java se corresponden con los nombres de fichero generados.

- Contar el número de estructuras semánticas o patrones que aparecen en un fichero para validar que se ha generado el número esperado de dichas estructuras en el código.

Como se puede ver, algunas de las características son más centradas a lenguajes de programación concretos (Java en este caso), y otras, como la última, son independientes del lenguaje de programación generado por el generador de código.

Para satisfacer estas características, el DSL que se explica en los siguientes apartados, proporciona varias construcciones que se muestran a modo de ejemplo a continuación (correspondientes al generador de máquina de estado UML a Java):

```
Validador ClaseJava
Carpeta cafsignalling {
        Carpeta componente {
                Carpeta java {
                        Carpeta maquinaestados {
                                Carpeta acciones {
                                        ForAll FunctionBehavior JavaFile;
                                }
                                Carpeta eventos {
                                        ForAll SignalEvent JavaFile;
                                }
                                Carpeta guardas {
                                        ForAll OpaqueExpression groupBy name JavaFile;
                                }
                        }
                }
        }
}
```

Como se puede observar a simple vista, el DSL permite mantener la jerarquía de directorios que impone el generador de código que se esté validando. Mediante el uso de instrucciones del tipo "**ForAll** FunctionBehavior **JavaFile**" se da soporte al desarrollador a especificar las características que desea validar sobre su código. Esta instrucción, en concreto, permite decirle al intérprete que se explica más abajo, que chequee que para todo elemento de tipo *FunctionBehavior* existe un fichero Java con el nombre del elemento.

4.2.3 FASE 2: DISEÑO DEL LENGUAJE

Partiendo de la base de que el entorno de validación va a ser universal y de que cada generador de código es distinto del resto, puesto que trata y resuelve un problema distinto, se hace necesario un mecanismo que permita amoldar el entorno de validación al generador sobre el que se realiza la validación.

Como solución a este planteamiento se decidió crear un DSL que permitiera a los desarrolladores especificar qué validaciones deseaban realizar. El DSL estará formado por un metamodelo, una gramática y un intérprete de esa gramática.

Durante la introducción del proyecto se presentó el concepto de lenguaje específico de dominio (DSL). Como en ella se dijo, un DSL es un lenguaje de dedicado a resolver un problema en un dominio en particular que da solución a problemas concretos y, por tanto, su alcance es muy limitado. Sin embargo, el hecho de que tenga alcance limitado no es un inconveniente, sino más bien una ventaja. Aunque un DSL puede ser muy complejo de elaborar, aporta grandes beneficios como mejora en la calidad, productividad, mantenimiento y portabilidad. Además permite validaciones a nivel de dominio, es decir, mientras las construcciones del lenguaje estén correctas, cualquier sentencia escrita puede considerarse válida. Incluso creando una gramática sencilla para el DSL se consigue facilitar tareas complejas.

En este proyecto el dominio es el entorno de validación y se hacía necesario un mecanismo de especificación de acciones a validar. Para permitir al desarrollador del generador de código especificar las validaciones que desea realizar sobre su código, se decidió crear una gramática sencilla e intuitiva, que además permitiera mantener y reflejar la estructura que posee el código generado.

Para ello, el primer paso es crear un metamodelo con los conceptos del lenguaje. Un modelo conforme a ese metamodelo (una instancia del metamodelo) sería una representación en memoria de la especificación. A través de este modelo, el intérprete se encarga de realizar las comprobaciones pertinentes.

A continuación se explican en detalle tanto el metamodelo ideado como el lenguaje creado:

4.2.3.1. DISEÑO DEL METAMODELO

Como se ha hecho referencia recientemente, se hace necesaria una forma de recoger las especificaciones de validación para pasárselas al intérprete. La solución adoptada para esta necesidad fue crear un metamodelo para continuar con la mecánica de modelos adoptada como solución global.

De este modo se diseñó el metamodelo Validador. En la siguiente figura se muestra el diagrama de clases que se diseñó para su posterior implementación utilizando EMF como *framework* de metamodelado:

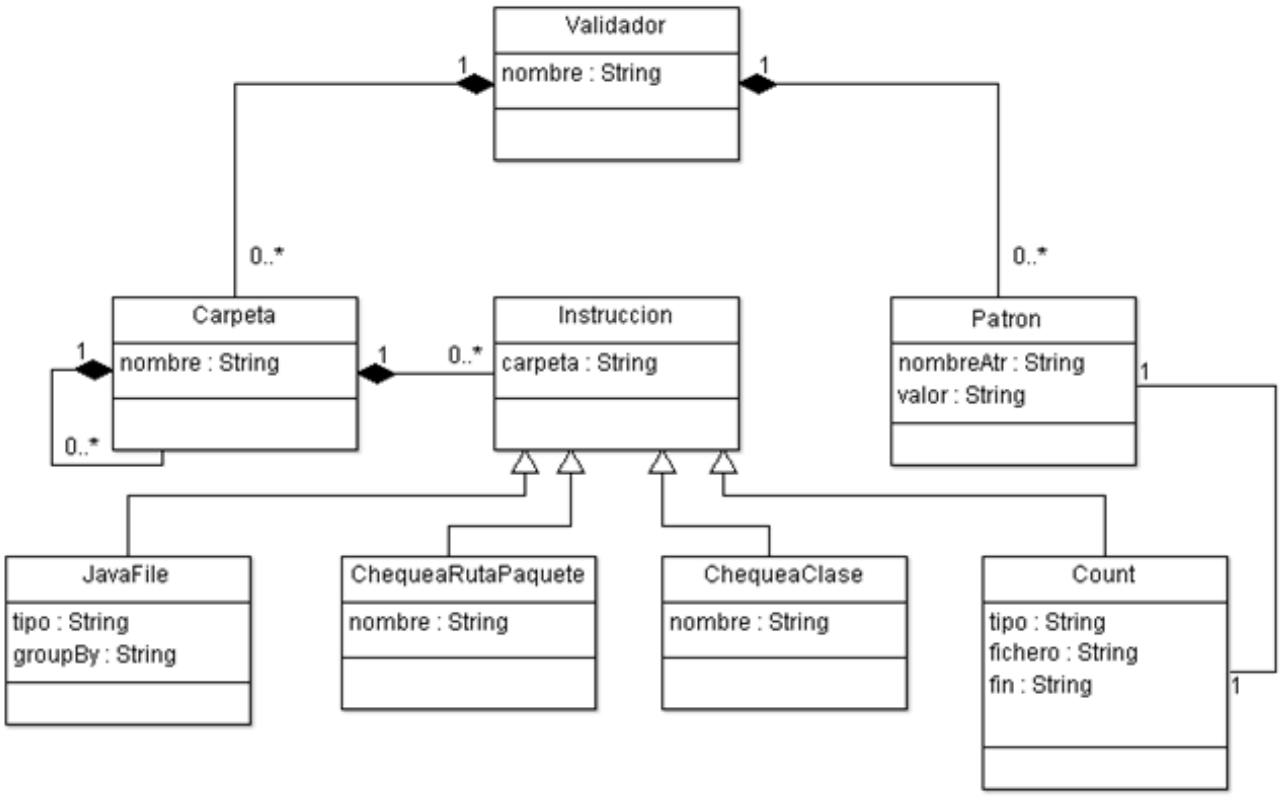


Figura 5. Metamodelo del validador

Puesto que normalmente el código generado por los generadores se almacenará en una carpeta y la mayoría de veces habrá una jerarquía de directorios, el metamodelo ha sido diseñado para poder mantener dicha jerarquía durante la validación y permitir realizar las comprobaciones por carpetas (meta-clase *Carpeta*, que pueden estar anidadas). De este modo cada carpeta que se especifique en el fichero de validación podrá contener cualquier instrucción para validar el contenido que se ha generado en ella (meta-clase *Instrucción*, donde cada carpeta puede tener varias o ninguna instrucción para validar). Cada instrucción se encarga de ejecutar una acción de validación distinta y se corresponde con cada una de las características descritas en la fase I (cada una de las meta-clases *Javafile*, *ChequeaRutaPaquete*, *ChequeaClase*, *Count*, se corresponde con una acción de validación).

Además el validador podrá contener patrones o expresiones regulares (meta-clase *Patron*) que se almacenarán en forma de *String* y serán definidos al comienzo del fichero de validación. Éstos permitirán especificar determinados patrones que se deseen identificar en el código generado, por ejemplo para contabilizar el número de apariciones de un tipo de función y validar que es el esperado.

Este diseño se corresponde con el prototipo de herramienta que se ha implementado, pero es extensible con toda la funcionalidad que se desee aportar en un futuro si se decide desarrollar la herramienta con más potencial.

4.2.3.2. SINTAXIS CONCRETA TEXTUAL

La sintaxis textual de la gramática creada se ha desarrollado mediante el uso del *framework Xtext* y permite especificar en un fichero las instrucciones de validación que se deben chequear sobre el código. Los ficheros de validación se deben crear con la extensión ".vl. Al estar implementado en Eclipse, se podría hacer un *plug-in* que permitiera la edición de ficheros de validación y de esta manera facilitar la tarea de especificación. Sin embargo, para el desarrollo del proyecto no se ha encapsulado está función en un *plug*-in. Más adelante, en el apartado de pruebas y resultados se verán ejemplos de especificaciones de validación.

En la **figura 6,** que se expone a continuación, se presenta la gramática y acto seguido se procede a la explicación de la conexión entre el metamodelo y dicha gramática y de la transformación que se produce del fichero de especificación al modelo que actúa como entrada al intérprete.

```
Validador returns Validador:
      'Validador' nombre=EString
              ( declaraciones+=Estructura (declaraciones+=Estructura)* )?
              ( carpetas+=Carpeta (carpetas+=Carpeta)* )?
      ;

Carpeta returns Carpeta:
      'Carpeta' nombre=EString'{'
            (instrucciones+=Instruccion ( instrucciones+=Instruccion)* )?
            (carpetas+=Carpeta ( carpetas+=Carpeta)* )?
      '}';

Instruccion returns Instruccion:
      JavaFile | NothingElse | ChequeaRutaPaquete | ChequeaClase | Count;

EString returns ecore::EString:
      STRING | ID;


Estructura returns Estructura:
      '#Estructura'nombreAtr=EString'=' valor=EString';';

JavaFile returns JavaFile:
      'ForAll' tipo=EString ('groupBy'groupBy=EString)?'JavaFile'';';

ChequeaRutaPaquete returns ChequeaRutaPaquete:
      'ForAll' nombre=EString'checkPackage'';';

ChequeaClase returns ChequeaClase:
      'ForAll' nombre=EString'checkClass'';';

Count returns Count:
      'ForAll' tipo=EString'count'eName=EString'in' fichero=EString
('endWith' fin=EString)?';';

NothingElse returns NothingElse:

      'NothingElse'';';
```

Figura 6. Gramática del DSL

Aunque la gramática es muy sencilla y la sintaxis de una gramática podría llegar a complicarse mucho y ser tediosa de elaborar, se consiguen varias ventajas y se da solución al problema que se planteaba de cómo permitir especificar validaciones personalizadas.

Como se puede ir viendo sin entrar en mucho detalle, la gramática permite establecer las validaciones siguiendo la jerarquía de carpetas que contenga el código generado. El validador contendrá una o varias carpetas de primer nivel, y cada una de ellas podrá contener carpetas de niveles más profundos en su interior.

A nivel de las carpetas raíz, también se podrán declarar patrones que como ya se mencionó anteriormente, servirán para especificar expresiones regulares útiles para validar determinadas características.

A través de las reglas definidas en la gramática se pueden realizar las especificaciones de validación. Una vez definidas por el desarrollador, éstas deben ser transformadas en modelo que será lo que reciba el intérprete para entender y ejecutar las acciones solicitadas. Para entender la gramática se explica cómo se relacionan las reglas de la misma con el metamodelo (descrito en la **figura 5**) para generar el modelo.

La regla principal que debe aparecer en cualquier fichero de validación es la siguiente:

```
Validador returns Validador:
      'Validador' nombre=EString
              ( declaraciones+=Estructura (declaraciones+=Estructura)* )?
              ( carpetas+=Carpeta (carpetas+=Carpeta)* )?
      ;
```

Esta regla permite hacer el mapeo al metamodelo siendo el nombre un *String* que da nombre al validador y deberá aparecer siempre. Las dos siguientes líneas, escritas entre paréntesis y con interrogación final indican que podrá o no haber elementos del tipo establecido en su interior. La primera línea indica que podrá haber una colección de elementos de tipo Estructura y la segunda una colección de elementos de tipo Carpeta. El símbolo "+=" permite especificar que se realiza una agregación a la colección del tipo que le siga. Por otro lado, el símbolo"*" permite indicar que podrá haber varios elementos del tipo contenido en el interior de sus paréntesis.

Como se puede observar en la estructura del metamodelo de la **figura 5** la clase Validador deberá tener un nombre, y podrá tener de 0 a varias estructuras y de 0 a varias carpetas.

La siguiente regla de la gramática es similar a la que se acaba de explicar, pero en esta ocasión expresa el elemento Carpeta. Igual que con el elemento Validador, cada carpeta deberá tener un nombre y podrá contener 0 o varias carpetas en su interior, y a diferencia que en la regla anterior, en este caso podrá contener 0 o varias instrucciones.

Las dos siguientes reglas que aparecen en la gramática siguen el siguiente patrón:

```
Instrucción returns Instruccion:
      JavaFile | NothingElse | ChequeaRutaPaquete | ChequeaClase | Count;
```

Estas reglas son reglas intermedias en la gramática que serían el caso análogo a una clase abstracta en Java cuyas clases que heredan de ella son las que se indican abajo. El operador "|" permite expresar que será de un tipo u otro, pero no de todos a la vez.

Finalmente, las siguientes reglas que hay siguen el siguiente patrón:

```
Count returns Count:
      'ForAll' tipo=EString'count'eName=EString'in' fichero=EString
('endWith' fin=EString)?';';
```

Con ellas se realiza el mapeo a los distintos tipos de instrucción definidos en el metamodelo. Éstas son las que expresan las distintas características que se permiten validar en el entorno hasta el momento.

En el siguiente apartado se explica qué acción de validación permite expresar cada una de estas últimas reglas definidas en la gramática.

4.3.3.4. RELACIONES REGLA-CARACTERÍSTICA

Durante la fase 1 se pensaron las características que el prototipo de herramienta iba a permitir validar con el fin de que se mostrara la potencialidad y utilidad de dicha herramienta para validación de generaciones de código. Para hacer de la herramienta un entorno genérico que permitiera validar distintos generadores de código, siendo el desarrollador el que especificara las validaciones deseadas, se creó la gramática

explicada anteriormente. Veamos ahora cómo la gramática permite expresar las distintas características que se fijaron como objetivo a implementar en el entorno.

La siguiente regla de la gramática permite validar que el número de ficheros que se generan para un determinado tipo de objeto, se corresponde con el número de ficheros real necesarios de dicho objeto:

```
JavaFile returns JavaFile:
      'ForAll' tipo=EString ('groupBy'groupBy=EString)?'JavaFile'';';
```

Para generadores que den como salida código Java y tengan asociada una clase por cada instancia de un determinado tipo de objeto, se deberá validar que existan tantos ficheros Java como instancias de ese tipo de objeto haya en el modelo.

La segunda característica que se proponía era chequear que la ruta de un fichero Java es correcta. Para ello se construyó la siguiente regla en la gramática:

```
ChequeaRutaPaquete returns ChequeaRutaPaquete:
      'ForAll' nombre=EString'checkPackage'';';
```

Puesto que el código que se genera puede contener una organización en directorios y subdirectorios, cuando se generan ficheros Java, era un error común que la ruta del paquete en el fichero no se correspondiera con la ruta real deseada. Para ello esta regla se encarga de validar que para todos los ficheros Java que se debieran crear, se comprobaran sus rutas.

Se modo similar ocurría con los nombres de las clases y los nombres de fichero, que no se correspondían. Para comprobar que las clases de los ficheros Java se corresponden con los nombres de fichero generados, que es la tercera característica que se fijó, se creó la siguiente regla en la gramática:

```
ChequeaClase returns ChequeaClase:

      'ForAll' nombre=EString'checkClass'';';
```

Por último se creó la siguiente regla en la gramática para contar el número de estructuras semánticas o patrones que aparecen en un fichero para validar que se ha generado el número esperado de dichas estructuras en el código.

```
Count returns Count:
      'ForAll' tipo=EString'count'eName=EString'in' fichero=EString
('endWith' fin=EString)?';';
```

Esta regla es aún más genérica que las anteriores, puesto que permite validar cualquier tipo de objeto, pero además es aplicable independientemente del lenguaje de programación que se utilice en el código de salida del generador.

En determinadas ocasiones, un cierto tipo de objeto en el modelo representa una estructura en el código resultante. Un ejemplo común, es que por cada tipo de objeto se deba crear una función o método. En el código generado deberá haber tantas funciones de ese tipo como objetos haya en el modelo. Para comprobar eso, el validador permite especificar estructuras que contengan, por ejemplo, patrones que permitan identificar dichas funciones. Mediante esta regla, se indica al intérprete que cuente el número de objetos que hay de ese tipo y compruebe en el código generado, en concreto en el fichero especificado, que se encuentran tantas funciones que sigan dicho patrón como objetos haya.

En la sección de pruebas y resultados se verán ejemplos más detallados de las especificaciones que se pueden realizar y se verá la utilidad que dichas reglas tienen sobre proyectos reales.

4.2.4. FASE 3: IMPLEMENTACIÓN DEL INTÉRPRETE

Como última fase en el entorno de validación, se debía implementar el intérprete que se encargara de evaluar las acciones especificadas mediante un programa escrito con el lenguaje presentado en las secciones anteriores, y ejecutar las acciones correspondientes para validar el código generado.

Se decidió implementar el intérprete en código Java en el entorno de Eclipse con idea de que también pudiera integrarse en forma de *plug-in*.

Éste toma como entradas el código generado, el modelo sobre el que se generó dicho código y el modelo de validación que contiene las especificaciones a validar realizadas por el desarrollador.

De forma recursiva se van recorriendo las carpetas desde las carpetas raíz hasta los niveles más profundos. Para cada una de ellas se recorren las instrucciones que contienen y se realizan las comprobaciones que imponen. Los errores que se van detectando se almacenan en una estructura creada para reportar errores. Al final de la validación se muestran como salida del validador un listado con todos los errores reportados, con detalles de donde se encuentran y a que se deben.

La principal dificultad del intérprete es que está diseñado para ser genérico y poder realizar validaciones sobre distintos generadores de código. Para ello debe acceder al modelo origen utilizando EMF Dinámico. Como entrada al intérprete se recibirá el modelo con el que se ha generado el código y su metamodelo correspondiente. Las funciones internas del intérprete se han desarrollado mediante el uso de métodos Java de una API de acceso a EMF proporcionada por el tutor, como son *allObjects()* para coger todos los objetos de un determinado tipo, y *getFeature()* para acceder a una característica concreta de ese tipo.

4.3. GENERADOR DE DOCUMENTACIÓN

Para dotar al proyecto de mayor valor, se decidió realizar una cuarta fase en la que se implementara un generador de documentación. Éste se encargaría de generar documentación sobre la jerarquía de módulos que componen un generador de código y la relación de llamadas entre reglas de cada módulo. Esta información se muestra en forma de grafos o diagramas y resultan muy útiles a la hora de mantener el código del generador.

Una vez más se decidió realizar la implementación en Eclipse y con idea de formar un *plug-in* con dicha funcionalidad.

Para generar la documentación relativa al generador, se decidió utilizar Graphviz. Como ya se expuso en el apartado de herramientas utilizadas, Graphviz es software de código abierto que permite representar y visualizar grafos. Mediante la

especificación de los grafos utilizando el sencillo lenguaje de Graphviz en un fichero de texto y ejecutando posteriormente el comando que genera la imagen final, se consigue presentar la información sobre el generador.

La aplicación del generador de documentación recibe como entrada el módulo inicial del generador de código y a partir de él realiza un recorrido por sus demás módulos. Conforme se recorre un módulo se escribe en el fichero de texto la sintaxis correspondiente para generar un nodo en el grafo. Del mismo modo ocurre cuando se recorren las distintas reglas de cada módulo.

Finalmente, se ejecuta un comando de llamada a Graphviz que se encarga de generar la imagen final a partir del fichero de texto creado.

A continuación se muestran los algoritmos aplicados para generar la documentación. Para el caso del grafo de dependencias entre módulos se diseñó el siguiente pseudo-código de algoritmo:

```
generaGrafoDependenciasModulos (Module modulo, HashSet<Module> visitados){
        Si modulo ya visitado {
                volver;
        }sino {
                añadeModuloAVisitados(modulo);
                creaNodoGrafoConModulo(fichero,modulo);
                Para todo auxModulo en modulo.imports {
                        creaConexionEntreNodos(modulo,auxModulo);
                }

                Para todo auxModulo en modulo.imports {
                        generaGrafoDependenciasModulos (auxModulo, visitados);
                }
        }
}
```

Para crear un nodo en el lenguaje de *Graphviz* se utiliza la siguiente sintaxis:

```
modulo.getName() + "[shape=box, label=\"<<module>>\n"

                                + modulo.getName() + ".mtl\"];"
```

En el fichero que contiene la especificación del grafo se crea por cada módulo una línea que contiene el nombre el módulo y los parámetros del nodo que se deseen. Estos parámetros permiten especificar la forma del nodo, el texto que aparecerá dentro

del nodo, el color y muchas otras características. Para este caso únicamente se ha especificado la forma del nodo mediante el parámetro "`shape=box`", que hace que el nodo tenga forma rectangular, y el texto que debe aparecer en el interior mediante el parámetro "`label=".`

Para crear las conexiones con otros módulos se genera una línea por cada módulo importado utilizando la siguiente sintaxis:

```
modulo.getName() + " -> " + importedModule.getName() + ";"
```

De esta manera se crea una arista entre el modulo y el módulo que importa en cada iteración.

Para el grafo de llamadas entre reglas se diseñó un algoritmo similar al anterior. A continuación se muestra el pseudo-código del algoritmo:

```
generaGrafoRelacionesReglas (Module modulo, HashSet<Module> visitados){
        Si modulo ya visitado {
                volver;
        }sino {
                añadeModuloAVisitados(modulo);

                Para toda plantilla en modulo {
                        Si plantilla es TemplateInvocation {
                                creaNodoGrafoConPlantilla(fichero,plantilla);
                                creaConexionConNodosConectados(plantilla);
                        }
                }

                Para todo auxModulo en modulo.imports {
                        generaGrafoRelacionesReglas (auxModulo, visitados);
                }
        }
}
```

Igual que en el caso del grafo de dependencias entre módulos, los nodos se crean utilizando la misma sintaxis, salvo que en esta ocasión el parámetro forma es del tipo "record". De este modo los nodos se representarán como cajas con dos secciones. Las conexiones entre nodos se realizan utilizando la misma sentencia.

En la siguiente sección de pruebas y resultados se mostrará la utilidad de la aplicación con ejemplos realizados sobre el proyecto de CAF Signalling.

5. PRUEBAS Y RESULTADOS

Durante el proceso de implementación del entorno de validación y el generador de documentación, se fueron realizando pruebas unitarias de cada una de las funciones conforme se iban implementando. Además se fueron realizando también pruebas de integración entre el intérprete y el fichero de especificación para validar el correcto funcionamiento durante su desarrollo.

Una vez se dio por finalizado el desarrollo, se realizaron pruebas de caja negra sobre todo el sistema para comprobar que la salida del validador era la esperada. Para ello se siguió el siguiente procedimiento:

- Paso 1: definir el alcance de la prueba.
- Paso 2: definir el resultado esperado a la salida del validador
- Paso3: realizar las modificaciones necesarias sobre el código generado con el fin de alterar su correcto funcionamiento y realizar las especificaciones de validación deseadas para validar el código.
- Paso 4: ejecutar el validador
- Paso 5: anotar los resultados de la prueba.

A continuación se detallan las pruebas realizadas y resultados obtenidos sobre el entorno de validación y sobre el generador de documentación. Estos fueron realizados sobre un proyecto real, el proyecto de CAF Signalling. En el proyecto se desarrollaron dos generadores de código para máquinas de estado UML, uno para generar código Java y otro para generar código C. De esta forma, las pruebas reflejan la aplicación universal de la herramienta a distintos generadores de código.

5.1. ENTORNO DE VALIDACIÓN

5.1.1. PRUEBA 1: FICHEROS JAVA

ALCANCE DE LA PRUEBA

Esta prueba tiene como objetivo validar que se ha generado un fichero por cada acción, evento y guarda existentes en el modelo que representa una máquina de estados, y además se mantiene la estructura de ficheros esperada.

RESULTADO ESPERADO

Para la prueba se modificará el generador para que se generen menos ficheros y el validador deberá detectar que no han sido generados.

Se espera como resultado que el generador informe de que hay ficheros que no han sido generados.

MODIFICACIONES EN EL GENERADOR Y FICHERO DE ESPECIFICACIÓN

En el modelo de pruebas hay 10 acciones (*FunctionBehavior*), 7 eventos (*SignalEvent*) y 1 guarda (*OpaqueExpression*). El código original generado contiene tantos ficheros Java como acciones, eventos y guardas aparecen en el modelo y estos se sitúan en sus carpetas correspondientes de acciones, eventos y guardas respectivamente.

Se ha modificado la regla del generador de código encargada de generar los ficheros Java por cada acción y guarda, de tal modo que sólo recorra las n-1 acciones/guardas para que el código resultante del generador contenga un fichero de acción y otro de evento menos de los esperados.

El fichero de especificación es el siguiente:

```
Validador ClaseJava
Carpeta cafsignalling {
        Carpeta componente {
                Carpeta java {
                        Carpeta maquinaestados {
                                Carpeta acciones {
                                        ForAll FunctionBehavior JavaFile;
                                }
                                Carpeta eventos {
                                        ForAll SignalEvent JavaFile;
                                }
                                Carpeta guardas {
                                        ForAll OpaqueExpression groupBy name JavaFile;
                                }
                        }
                }
        }
}
```

RESULTADO DE LA PRUEBA

Tras ejecutar el intérprete sobre el código que ha sido alterado se produce la siguiente salida:

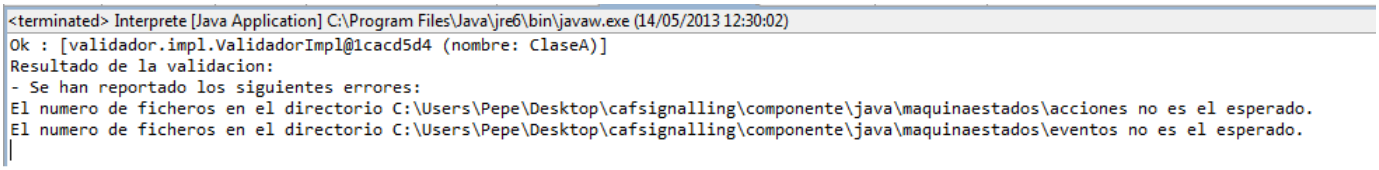


Figura 7. Salida prueba 1

Como se puede observar en la imagen, el intérprete ha detectado que el número de ficheros en los directorios acciones y eventos no es el esperado, tal y como se especificó que debía ser en el apartado de resultado esperado de la prueba.

Resultado de la prueba: **Satisfactorio**

5.1.2. PRUEBA 2: RUTA PAQUETE Y NOMBRE CLASES

ALCANCE DE LA PRUEBA

El objetivo de la prueba es validar que todas las rutas de paquete de los ficheros generados son correctas y además los nombres de clase se corresponden con los nombres de fichero.

RESULTADO ESPERADO

En esta prueba se espera que el validador detecte los ficheros que contienen su ruta de paquete errónea, así como también aquellos ficheros cuyo nombre de clase no se corresponda con el nombre de fichero.

El intérprete reportará los errores detectados indicando que ficheros son los que contienen el error, y en concreto, que error contiene.

MODIFICACIONES EN EL GENERADOR Y FICHERO DE ESPECIFICACIÓN

Para esta prueba se ha procedido a modificar el generador de tal modo que alterare el nombre del paquete uno de los ficheros. Además se ha modificado también para que cambie el nombre de uno de los ficheros.

El fichero de especificación para realizar la validación sobre estos cambios es el que sigue:

```
Validador ClaseJava
Carpeta cafsignalling {
        Carpeta componente {
                Carpeta java {
                        Carpeta maquinaestados {
                                Carpeta acciones{
                                        ForAll FunctionBehavior checkPackage;
                                        ForAll FunctionBehavior checkClass;
                                }
                        }
                }
        }
}
```

RESULTADO DE LA PRUEBA

Después de pasar el código alterado por el intérprete, éste muestra la siguiente salida:

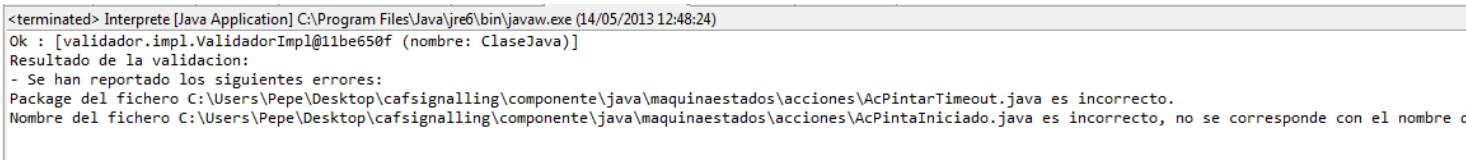


Figura 8. Salida prueba 2

Como era de esperar, el intérprete indica los errores detectados tal y como se preveía.

Resultado de la prueba: **Satisfactorio**

5.1.3. PRUEBA 3: FUNCIONES ESTADO

ALCANCE DE LA PRUEBA

En esta ocasión el objetivo de la prueba es validar que para cada estado que se encuentre en el modelo exista una función de estado. Se procede a realizar la prueba con un código mal generado en el que sólo haya 3 funciones de estado en el código, habiendo un total de 4 estados en el modelo; y sobre un código bien generado en el que estén las 4 funciones de estado que se espera que haya.

RESULTADO ESPERADO

En esta ocasión el intérprete deberá informar:

A. De que el número de ocurrencias del patrón especificado no es el correcto, para el caso del código mal generado.
B. De que el número de ocurrencias del patrón especificado es el correcto, para el caso del código bien generado.

MODIFICACIONES EN EL GENERADOR Y FICHERO DE ESPECIFICACIÓN

Para el caso A, se ha modificado el generador para que recorra sólo los n-1 estados que se encuentre primero. De esta forma el código generado contiene una función de estado menos de las esperadas.

Para el caso B, se genera el código con la versión original del generador, que genera una salida correcta.

Para ambos casos el fichero de especificación es el siguiente:

```
Validador ClaseA

#Patron C_Struct = "Boolean_teen_PDVL_funEstado(\\p{Alnum})*\\((\n.*){4}\\{";

Carpeta cafsignalling {
        Carpeta componente {
                Carpeta java {
                        ForAll State count C_Struct in "PDVL000.c";
                }
        }
}
```

Observación: La instrucción a validar se aplicará sobre el fichero *PDVL000.c* tal y como se indica en la propia instrucción. Además ésta indica que se quiere contar el número de apariciones de tipo *C_Struct*, que como aparece arriba, es una expresión regular. El intérprete deberá contabilizar el número de apariciones que sigan el patrón especificado en la estructura. En este caso, como se ha indicado en el alcance de la prueba, se desea detectar que exista una función de estado por cada estado en el modelo. Esto se consigue mediante la expresión regular especificada en *C_Struct.*

RESULTADO DE LA PRUEBA

Tras realizar la ejecución del intérprete sobre el código mal generado (en el que se había eliminado una de las 4 funciones de estado originales) se produce la siguiente salida:

```
<terminated> Interprete [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (14/05/2013 13:11:54)
Ok : [validador.impl.ValidadorImpl@4e823618 (nombre: ClaseA)]
Resultado de la validacion:
- Se han reportado los siguientes errores:
El numero de estructuras C_Struct es incorrecto, no se corresponde con el numero esperado del modelo.
```

Figura 9. Salida prueba 3A

Al obtener esta salida, el programador que realiza las especificaciones de validación sabe que el código es incorrecto y que hay funciones de estado que no se han generado. Esto le ayudará a detectar el posible error sobre el modelo o sobre el generador de código.

Una vez incluida de vuelta la función que se eliminó y tras ejecutar el validador sobre el código correcto se obtiene la siguiente salida:

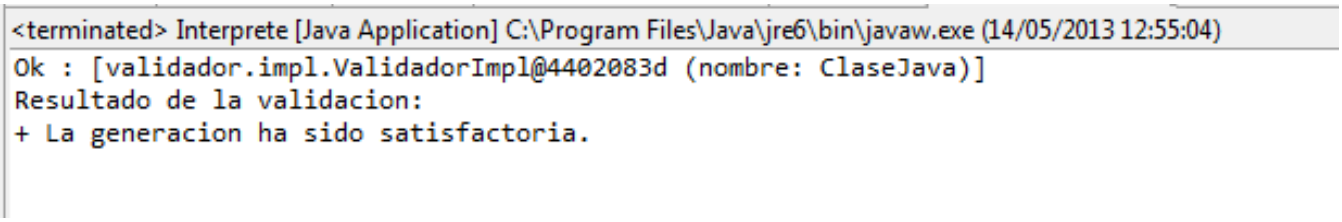


Figura 10. Salida prueba 3B

Resultado de la prueba: **Satisfactorio**

5.1.4. PRUEBA 4: CÓDIGO CORRECTO

ALCANCE DE LA PRUEBA

Esta prueba tiene como objetivo realizar la validación sobre un código generado correctamente y calificado como válido de manera manual por un programador. La prueba se realiza sobre el generador de código Java, en el que se aplicarán todas las validaciones posibles que permite el prototipo del entorno de validación hasta ahora.

RESULTADO ESPERADO

El intérprete deberá informar al usuario de que la validación ha sido satisfactoria y no se han detectado errores sobre el código.

MODIFICACIONES SOBRE EL GENERADOR Y FICHERO DE ESPECIFICACIÓN

En esta ocasión se parte de un código bien generado y validado manualmente y, por tanto, no procede ninguna modificación sobre el generador.

El fichero de especificación para aplicar todas las validaciones es el siguiente:

```
Validador ClaseJava
Carpeta cafsignalling {
        Carpeta componente {
                Carpeta java {
                        Carpeta maquinaestados {
                                Carpeta acciones {
                                        ForAll FunctionBehavior JavaFile;
                                        ForAll FunctionBehavior checkPackage;
                                        ForAll FunctionBehavior checkClass;

                                }
                                Carpeta eventos {
                                        ForAll SignalEven tJavaFile;
                                        ForAll SignalEvent checkPackage;
                                        ForAll SignalEvent checkClass;

                                }
                                Carpeta guardas {
                                        ForAll OpaqueExpression groupBy name JavaFile;
                                        ForAll OpaqueExpression checkPackage;
                                        ForAll OpaqueExpression checkClass;

                                }
                        }
                }
        }
}
```

RESULTADO DE LA PRUEBA

Como era de esperar, la validación produce el siguiente resultado tras ejecutar el intérprete sobre el código válido:

```
<terminated> Interprete [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (14/05/2013 12:55:04)
Ok : [validador.impl.ValidadorImpl@4402083d (nombre: ClaseJava)]
Resultado de la validacion:
+ La generacion ha sido satisfactoria.
```

Figura 11. Salida prueba 4

Resultado de la prueba: **Satisfactorio**

5.2. GENERADOR DE DOCUMENTACIÓN

5.2.1. EJEMPLO DE GENERACIÓN DE DOCUMENTACIÓN

El objetivo de este ejemplo es mostrar la funcionalidad y utilidad de generar la documentación en forma de grafo de dependencias entre módulos y de relación de llamadas entre las distintas reglas que componen cada módulo para un generador de código. En concreto se ha realizado la generación de documentación para el generador de C desarrollado en el proyecto de CAF Signalling.

Tras realizar la ejecución del generador de documentación pasándole como entrada el módulo inicial del generador (*Paquete.mtl*), se generan las dos imágenes que contienen los grafos correspondientes a las dependencias entre módulos y relaciones entre reglas que se deseaban obtener. Además se generan ficheros intermedios que contienen el código en formato *Graphviz* que dan lugar a dichas imágenes. No obstante, estos no son relevantes para la documentación por lo que no se muestran como resultado de la prueba.

A continuación se muestran los grafos generados mediante el generador:

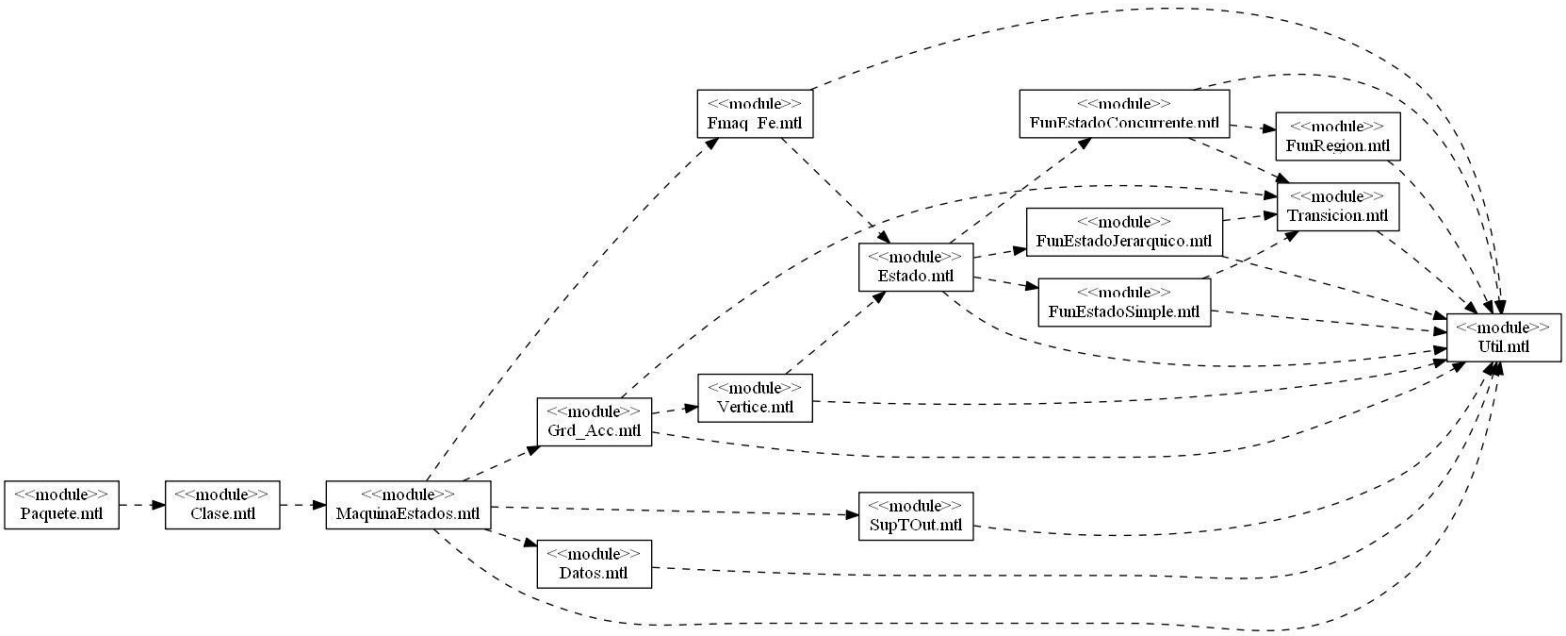


Figura 12. Grafo de dependencias entre módulos

La imagen anterior muestra cómo están relacionados los módulos que componen el generador de código C. Esta información resulta muy útil para documentar el generador desarrollado para la empresa. Actualmente esta imagen forma parte de la documentación realizada para la empresa sobre el generador de código y permitirá facilitar las tareas de mantenimiento a los desarrolladores que finalmente se encarguen de mantener el generador.

Del mismo modo, también se incluye en la documentación el siguiente grafo de dependencias entre reglas:

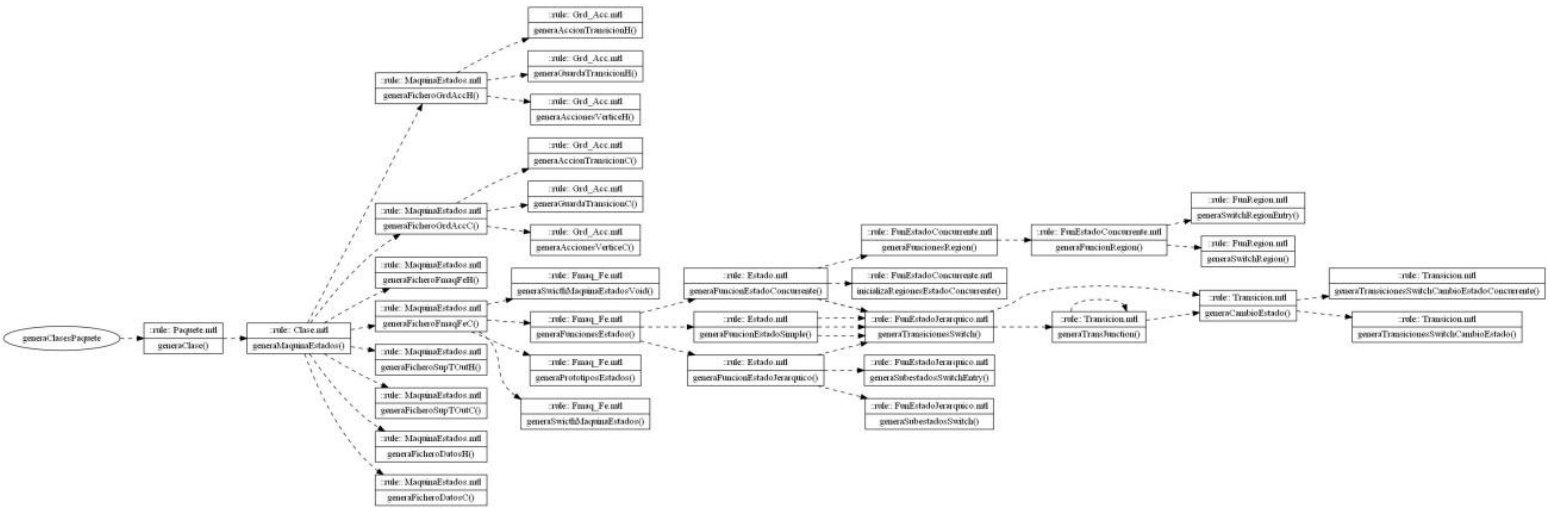


Figura 13. Grafo de dependencias entre reglas

.

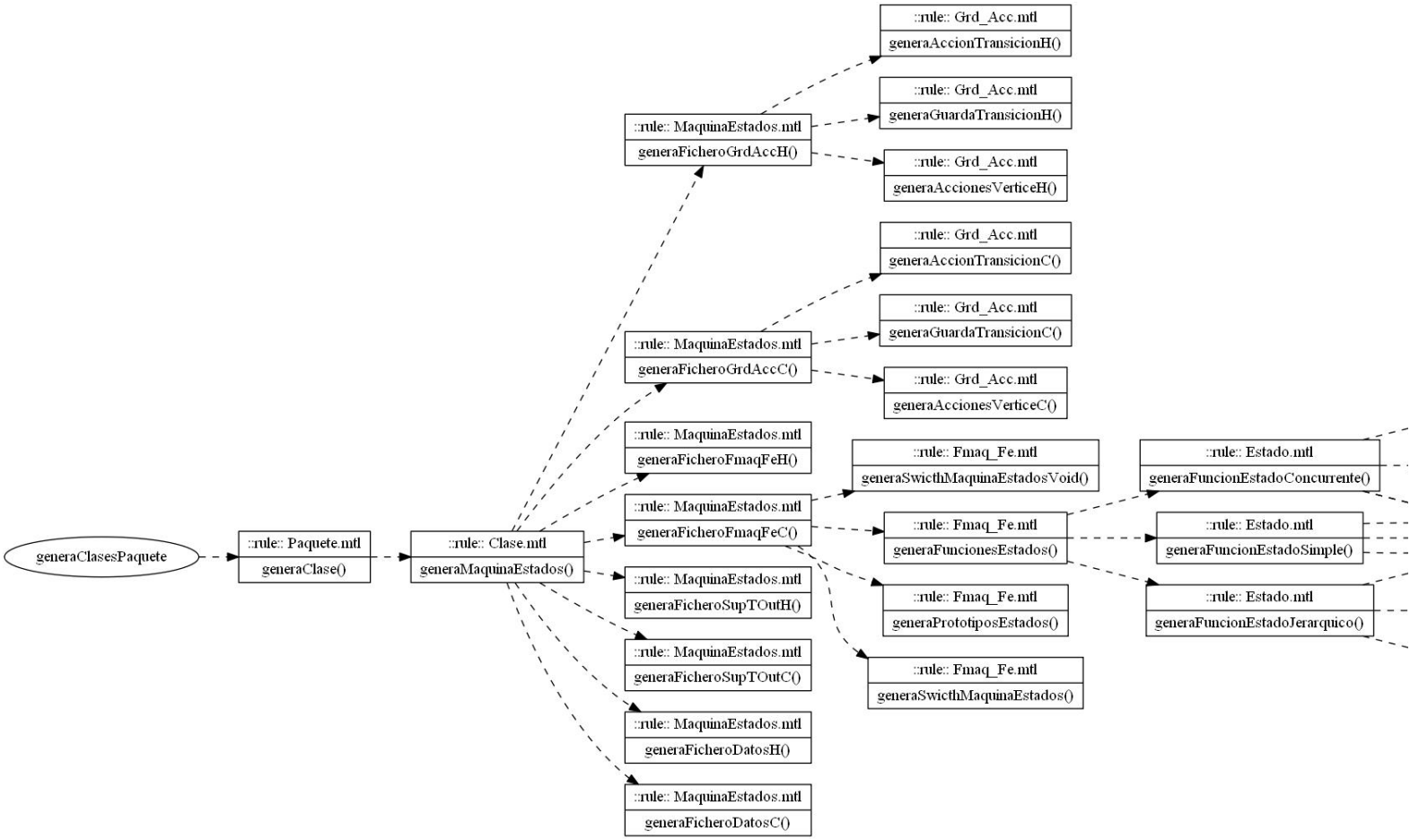


Figura 14. Parte del grafo aumentada de dependencias entre reglas

En la imagen anterior, que se corresponde con una parte aumentada del grafo de dependencias entre reglas del generador de código C, se puede observar cómo desde el módulo *Paquete.mtl* se realiza una llamada a la regla *generaClase().* Mediante la llamada a esta regla, se accede al módulo *Clase.mtl*. Cada módulo generará el código correspondiente accediendo a cada una de las reglas que contiene según sean llamadas desde otros módulos.

6. CONCLUSIONES Y TRABAJO FUTURO

Son muchos los beneficios que aportan los metamodelos y modelos en el desarrollo software como se ha hecho referencia en este trabajo. Nos encontramos en transición hacia la industrialización y automatización de los procesos de desarrollo de software. La rápida evolución de éste y la cantidad de líneas de investigación abiertas dirigidas al estudio del Desarrollo de Software Dirigido por Modelos, junto con los grandes beneficios que aportan estos, hacen prever una nueva revolución en técnicas de desarrollo en un futuro no muy lejano, aunque aún queda mucho camino por recorrer.

Con la aparición del paradigma del DSDM surge la generación de código automática a partir de modelos, que trae consigo una nueva necesidad. Los procesos automatizados para la generación automática requieren de nuevas técnicas que validen y garanticen la correcta que el resultado de la generación es válido y correcto desde el punto de vista funcional.

Como se ha reflejado en este trabajo, resulta muy útil para las empresas poder validar determinados requisitos sobre el código que generan sus generadores de código. Por ello resulta interesante idear y desarrollar una herramienta universal que permita realizar validaciones personalizadas sobre cualquier generador. Igualmente es importante poder documentar y visualizar de manera sencilla la organización de un generador. En este trabajo se ha realizado prototipos de estas dos herramientas que permiten mostrar la potencialidad que puede ofrecer al sector dentro del paradigma del DSDM y que han mostrado ser útiles al ser aplicadas a un proyecto real en una empresa.

Como trabajo futuro existe la posibilidad de integrar los dos prototipos de herramientas en forma de *plug-in* para Eclipse. Además se podría seguir extendiendo el lenguaje de validación para soportar otras construcciones y poder validar nuevas características e incluso, poder realizar compilaciones del código resultante tras ser validado dando como salida un informe con los errores de compilación. Estos servirían para detectar los posibles errores que contuviera el generador de código.

También resultaría interesante añadir al entorno de validación la posibilidad de comparar las salidas de diferentes versiones del generador de código para detectar posibles problemas y errores que se introdujeran al aplicar mejoras en el generador.

7. GLOSARIO

- **Metamodelo**: es el elemento que representa las reglas de construcción de un lenguaje, es decir, la sintaxis abstracta del lenguaje

- **Modelo**: es la descripción de un sistema bajo estudio utilizando un lenguaje, que vendrá definido por un metamodelo.

- **Lenguaje específico de dominio (DSL)**: es un lenguaje de dedicado a un problema de dominio en particular y que da solución a un problema concreto. Está compuesto por un metamodelo, una sintaxis textual y un intérprete que relaciones el metamodelo con la sintaxis textual.

- **Transformación de modelos**: consiste en la conversión de un modelo a otro modelo siguiendo un metamodelo distinto (transformación modelo-a-modelo); o la conversión modelo a texto siguiendo unas reglas para generar código de manera automática (transformación modelo-a-texto).

- **Generador de código**: software desarrollado para generar código de manera automática a partir de modelos. Es una transformación de tipo modelo-a-texto.

- **Sintaxis concreta**: es una representación de modelos en forma entendible y manejable por un usuario. Ejemplos de sintaxis concreta pueden ser una sintaxis textual o una sintaxis gráfica.

8. BIBLIOGRAFÍA Y REFERENCIAS

[1] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. 2006. Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons

[2] UML: http://www.uml.org/

[3] Martin Fowler. 2010. Domain Specific Languages (1st ed.). Addison-Wesley Professional

[5] MOF-to-Text: http://www.omg.org/spec/MOFM2T/1.0/

[6] Stuermer, I., Conrad, M., Doerr, H., & Pepper, P. (2007). Systematic testing of model-based code generators. Software Engineering, IEEE Transactions on, 33(9), 622-634

[7] Stürmer, Ingo, and Mirko Conrad. "Code Generator Testing in Practice." GI Jahrestagung (2) (2004): 33-37

[8] Stürmer, Ingo, Daniela Weinberg, and Mirko Conrad. "Overview of existing safeguarding techniques for automatically generated code." ACM SIGSOFT Software Engineering Notes. Vol. 30. No. 4. ACM, 2005

[9] Stürmer, Ingo, Daniela Weinberg, and Mirko Conrad. "Overview of existing safeguarding techniques for automatically generated code." ACM SIGSOFT Software Engineering Notes. Vol. 30. No. 4. ACM, 2005

[10] García-Domínguez, A., Kolovos, D. S., Rose, L. M., Paige, R. F., & Medina-Bulo, I. (2011). EUnit: a unit testing framework for model management tasks. In Model Driven Engineering Languages and Systems (pp. 395-409). Springer Berlin Heidelberg

[11] Lin, Y., Zhang, J., Gray, J.: A testing framework for model transformations. In: Beydeda, S., Book, M., Gruhn, V. (eds.) Model-Driven Software Development, pp. 219–236. Springer-Verlag, Berlin, Germany (2005)

[12] Guerra, E., de Lara, J., Kolovos, D. S., Paige, R. F., & dos Santos, O. M. (2011). Engineering model transformations with transML. Software & Systems Modeling, 1-23

[13] Acceleo: http://www.eclipse.org/acceleo/

[14] OMG: http://www.omg.org/

[15] Xtext: http:// www.eclipse.org/Xtext/

[16] Eclipse: http://www.eclipse.org/

[17] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. Emf: Eclipse Modeling Framework 2.0 (2nd ed.). Addison-Wesley Professional