UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO

Un entorno de generación automática de modelos de prueba

Autor: Juan Ignacio Cornet Recchimuzzi

Tutora: Esther Guerra

Mayo-Junio 2013

RESUMEN

La propuesta de realizar un entorno de generación automática de modelos de prueba de modelos surge a raíz de un generador de código que trabaja con modelos como parámetro de entrada. Dicho generador recibía un modelo de máquina de estados y generaba, a partir del mismo, el código C equivalente a la misma. Al intentar realizar las pruebas de validación del generador de código, se plantea el problema de que generar modelos válidos a mano para el mismo es un trabajo bastante laborioso, y que valdría la pena automatizar. Con esta premisa, se plantea el generador de modelos como un plug-in para Eclipse que trabaje de forma genérica con cualquier meta-modelo.

Observando las otras herramientas similares que existen en la actualidad, se perciben dos problemas fundamentales: generalmente las herramientas no trabajan directamente sobre meta-modelos Ecore, sino que traducen el mismo a un formato interno (lo cual es costoso); y no es posible "controlar" la generación de modelos definiendo diferentes estrategias (como puede ser creación mínima de objetos, permitir al generador completar los modelos automáticamente con clases que no estén definidas en las restricciones de entrada, etc.)

Con estas limitaciones, se decide realizar el generador utilizando EMF para trabajar con cualquier meta-modelo Ecore, ya que ofrece una librería reflexiva que lo permite. Además, en el análisis de diseño, se plantean como requisitos poder definir distintas estrategias de generación, como las anteriormente citadas.

El documento expuesto a continuación recoge, además de una breve contextualización del proyecto y las tecnologías utilizadas, el análisis del generador de modelos, el diseño de su arquitectura, el algoritmo de generación implementado y muestras de su uso tanto con un meta-modelo de prueba como con el meta-modelo utilizado en el generador de código C.

SUMMARY

The idea of developing an automatic model generation environment comes from a code generator that works with models as its input. This code generator received a state machine model and then it generated the equivalent C code for that state machine. When trying to perform validation tests of the code generator one problem arises: the generation of valid models for it by hand is a hard and time consuming task which could be automated. Based on this premise, the model generator is a solution presented as an Eclipse plug-in. This plug-in will work in a generic way with every meta-model.

After researching on similar tools which are currently existing, two main problems can be identified: generally these tools do not work directly on Ecore meta-models but they translate it to an internal format (poor performance); and it is not possible to "control" the model generation with these tools establishing different strategies (like minimum object creation, allowing the generator to complete the models automatically with classes that are not defined in the input restrictions, etc.)

With these limitations, it is decided to carry out the generator using EMF in order to work with any ECORE meta-model as it offers a reflexive library which makes it possible. Moreover, in the design analysis, it is suggested as a requirement the possibility to define different generation strategies, like the ones mentioned before.

This paper contains a brief contextualization with the project itself and the used technologies, the analysis of the model generator, the design of its architecture and the implemented generation algorithm with samples of its use with a test meta-model and with the meta-model used in the C-code generator.

PALABRAS CLAVE

La siguiente lista contiene las cinco palabras más descriptivas que podrían utilizarse para catalogar al proyecto:

- Desarrollo dirigido por modelos
- Pruebas de software
- Meta-modelado
- Modelado de software
- Generación de código

KEYWORDS

The following list contains the five most descriptive words which could be used to catalog the project:

- Model driven development
- Software testing
- Meta-modeling
- Software modeling
- Code generator

ÍNDICE

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

La realización del generador de modelos genérico viene motivada como solución a un problema encontrado en un entorno de trabajo real. Trabajando en el desarrollo de un generador de código C para máquinas de estados a partir de un modelo de entrada UML [1], se encuentra el problema que realizar casos de prueba para comprobar el correcto funcionamiento del mismo es un trabajo laborioso. Sería ideal, de alguna forma, poder automatizar lo máximo posible este proceso.

Con este planteamiento inicial, se propone la idea de realizar un plug-in para Eclipse que, dado unos parámetros de entrada concretos, genere una serie de modelos válidos que sirvan para ser utilizados como *input* del generador de código. Así, solucionaríamos el problema anterior, pero, ¿se podría mejorar la solución? Hacer un generador para cada proyecto, ¿no sería demasiado laborioso? Las respuestas a ambas preguntas es sí, por lo que el generador, para que sea viable, es decir, que el tiempo invertido en realizarlo sea inferior al tiempo que llevaría hacer el trabajo sin el mismo, ha de ser genérico.

Las características de este generador (además de trabajar con cualquier meta-modelo) deben poder permitir limitar la creación de objetos en el modelo (definir un ámbito de búsqueda), restringir el número mínimo y máximo de objetos de cada tipo en los modelos generados, así como de sus conexiones; poder seleccionar la estrategia de generación de los modelos entre un conjunto existente, como por ejemplo, generar el modelo de menor tamaño que sea conforme al meta-modelo y esté dentro del ámbito de búsqueda, crear objetos de tipos no especificados en el ámbito de búsqueda si son necesarios para crear un modelo válido, o elegir heurísticas a la hora de conectar los objetos del modelo.

El documento expuesto a continuación tiene como motivo ilustrar los detalles de la realización y el funcionamiento del generador. Para ello, se dividirá la estructura del mismo de la siguiente forma:

- Una primera parte de introducción y contexto de desarrollo del proyecto, secciones 1 y 2, en las que se pondrá al lector en situación de qué es lo que se ha hecho, por qué y para qué.
- Una segunda parte que consiste en la explicación formal de cómo se ha realizado el proyecto (el análisis, diseño e implementación del mismo), qué tecnologías se han utilizado y qué otros proyectos o programas similares hay a día de hoy, secciones 3, 4, 5 y 6.
- Una parte final en la que se mostrará el programa en funcionamiento y cómo responde a distintos casos de entrada, así como las conclusiones del proyecto y el posible desarrollo futuro que pueda tener, secciones 7 y 8.

Además de la estructura comentada anteriormente, al final del documento se podrá encontrar un glosario y una lista de referencias utilizadas.

2. CONTEXTO DE DESARROLLO

El contexto de desarrollo se engloba en un proyecto realizado para *cafsignalling* [2]. Para dicho proyecto, se ha pedido desarrollar un generador de código C que dado un modelo UML correspondiente a una máquina de estados, genere los ficheros necesarios que representan la misma en código C. El lenguaje utilizado para el desarrollo del mismo ha sido Acceleo.

Sin entrar más en detalle, veremos este generador como una "caja negra" a la que si le pasamos como parámetro un modelo, la misma se encargará, aprovechando las utilidades de Acceleo, de realizar ciertas transformaciones para acabar generando los archivos .c y .h representativos del modelo dado.

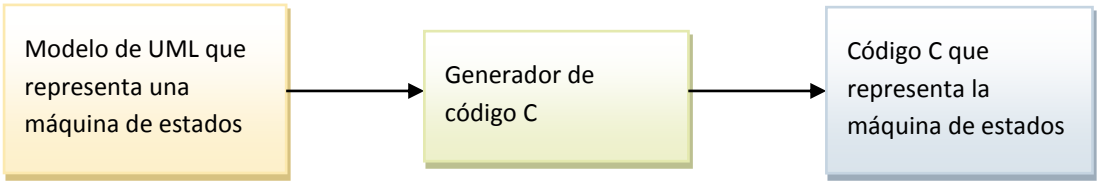

Figura 1. Arquitectura del generador de código C.

Partiendo de estas condiciones, surge la idea de realizar un programa que sirva como generador de casos de prueba. Más concretamente, se necesita realizar un programa que cree distintos modelos válidos para poder probar el generador de código desarrollado. Si extrapolamos la idea de este proyecto particular a cualquier proyecto que trabaje con modelos, finalmente lo que necesitamos es un generador de modelos genérico que genere modelos válidos para cualquier meta-modelo dado.

2.1 DSDM

El proyecto de generación a partir de modelos UML, del cual surge la iniciativa del generador de modelos, no es más que un ejemplo de DSDM (Desarrollo de Software Dirigido por Modelos).

El DSDM es un nuevo paradigma para el desarrollo de software en el que los modelos son los principales elementos de desarrollo. El objetivo del mismo es principalmente la generación de código a partir de ciertas especificaciones o modelos expresados mediante lenguajes de modelado específicos de dominio (DSLs, *Domain Specific Language*) o lenguajes de modelado de propósito general (como UML). Estos lenguajes proporcionan un nivel mayor de abstracción, así como un lenguaje de alto nivel como Java, proporciona mayor nivel de abstracción que código en ensamblador.

Debido a sus diferentes ventajas (permite desarrollar software de mayor calidad más rápido, se eleva el nivel de abstracción, se utiliza la documentación -los modelos son normalmente parte de la documentación de un proyecto de software- para realizar la implementación, etc.), podemos decir que el futuro de la programación se enfocará principalmente a esta nueva metodología que se expande año tras año.

3. ESTUDIO DEL ESTADO DEL ARTE

3.1 ENFOQUES EXISTENTES PARA LA GENERACIÓN DE MODELOS

A la hora de generar modelos, hay dos enfoques principales: usar gramáticas de grafos o usar *constraint solvers*. En el primer enfoque, se tiene una gramática generativa que contiene reglas cuya aplicación van creando los diferentes elementos del modelo. Se tiene que construir una gramática por cada meta-modelo para el que se quieran generar modelos. En la actualidad, no hay ninguna herramienta que automatice la creación de estas gramáticas para meta-modelos Ecore.

El segundo enfoque es el más extendido actualmente. En éste, la entrada está formada por el meta-modelo de entrada, un conjunto de restricciones que indican características adicionales de los modelos generados (más allá de la estructura dada por el meta-modelo) y un *scope*, que fija el espacio de búsqueda (número mínimo y máximo de objetos de cada tipo y valores permitidos para los tipos de datos). A veces, también es posible dar como entrada un modelo semilla. La búsqueda de modelos se hace en dos pasos: primero la entrada se traduce al formato interno usado por el *constraint solver* concreto (ej. *boolean satisfiability* [3], SMT [4], CSP [5]), y a continuación, se realiza la búsqueda. El principal problema es que estas traducciones son complejas, y muchas veces, el tiempo de traducir la entrada al formato interno es elevado, incluso más que el tiempo de búsqueda en sí. Debido a esto, se ha determinado que una de las características del generador realizado (que seguirá este enfoque) sea que trabaje directamente con meta-modelos Ecore, sin realizar ninguna traducción intermedia.

La mayoría de generadores de modelos construidos sobre *constraint solvers* trabajan sobre UML [6]. También hay una herramienta llamada EmfToCSP [7] que admite como entrada un meta-modelo Ecore, que se traduce internamente a CSP para la generación de modelos. En esta herramienta, además, es posible indicar restricciones adicionales de los modelos a generar usando el lenguaje OCL.

Por último, respecto a los criterios de generación de modelos:

- En [8,9] se generan modelos a partir del meta-modelo y de unos criterios de cobertura del meta-modelo, como partición de los valores de los atributos o número de clases
- En [10] se usa un lenguaje imperativo para crear modelos. El lenguaje permite dar valores aleatorios a los atributos de entre un conjunto de valores posibles, y también puede ser aleatoria la asignación de objetos a referencias.
- En [6,8,11] se pueden dar modelos semilla.

La principal diferencia entre las herramientas existentes y el generador realizado es la posibilidad de definir distintas estrategias de generación (ver punto 4). Además, este último trabaja directamente con meta-modelos Ecore y genera directamente modelos XMI, mientras que con las otras herramientas, como ya se ha comentado, se hace una traducción a un formato interno, y además habría que *parsear* la salida si quisiéramos tenerla en formato XMI.

3.2 TECNOLOGÍAS UTILIZADAS

3.2.1 ACCELEO

Se ha utilizado principalmente a la hora de desarrollar el generador de código para el cual se ha desarrollado el generador de modelos.

Acceleo [12] es un lenguaje y un IDE de generación de código para Eclipse (un plug-in para el mismo de código abierto) que implementa el estándar MOFM2T [13] del OMG [14]. El IDE cuenta con varias características, como una sintaxis simple, generación eficiente, etc.

3.2.2 ECLIPSE

Eclipse [15] es un programa de código abierto compuesto por un conjunto de herramientas de programación. Normalmente se ha utilizado para desarrollar diferentes IDEs, como el de Java (*Java Development Tools,* JDT) o el de Acceleo comentado en el punto anterior.

Debido al diseño de su arquitectura, es muy fácil añadirle nueva funcionalidad simplemente desarrollando un plug-in para el mismo (de hecho, Eclipse proporciona la base, luego la mayor parte de sus funcionalidades vienen como plug-ins que el usuario puede descargarse e incorporar).

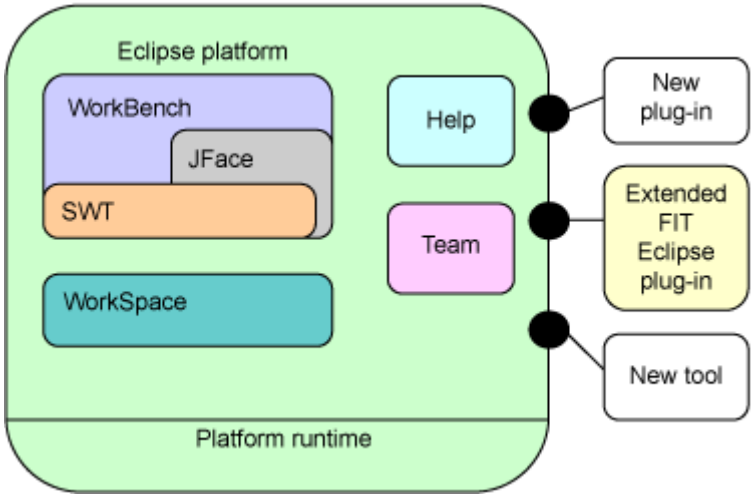

Figura 2. Arquitectura de Eclipse.

3.2.3 EMF

EMF, como su nombre indica (*Eclipse Modeling Framework* [16,17]*)*, es un *framework* de modelado que explota las facilidades de Eclipse (se integra al mismo como un plug-in). Permite sincronizar el modelo de una aplicación (expresada en UML), con su implementación (en Java) y el mecanismo de persistencia (XML). Para ello proporciona herramientas para generar automáticamente cualquiera de estas 3 vistas de una aplicación a partir de las otras (ej. generar código Java a partir de un modelo UML).

Lo interesante de EMF para el generador desarrollado es que provee una API reflexiva que nos permite tratar los datos de forma genérica, sin saber a qué meta-modelo en concreto pertenecen. Debido a esto, podemos realizar un generador genérico que se pueda utilizar en cualquier proyecto, sin importar que no usen los mismos meta-modelos (un aspecto fundamental del mismo).

3.2.4 XTEXT

XText [18] es un *framework* de código abierto para Eclipse para el desarrollo de lenguajes de programación y lenguajes de dominio específico (DSLs, *Domain Specific Languages)*.

Cubre todos los aspectos de la infraestructura de un lenguaje (o gramática) completo: intérpretes, *parsers*, etc., además de ofrecer funcionalidad de utilidad como colorear las palabras reservadas del lenguaje (“coloreo de sintaxis”), auto completar, etc.

4. ANÁLISIS

Como ya se ha contado a lo largo de la introducción y el contexto de desarrollo, el principal objetivo del generador es servir como recurso para las pruebas de software que está enfocado en la utilización de modelos.

Debido a que cada proyecto es distinto, se realizará un generador que sea flexible y tenga parámetros de configuración que le permitan ser de utilidad en diferentes contextos. Para lograr esta flexibilidad, se permite la configuración de dos aspectos de la generación de modelos: limitaciones concretas sobres los objetos creados y sus conexiones, y especificaciones para definir la forma de trabajar del generador.

4.1 ESPACIO DE BÚSQUEDA

4.1.1 NÚMERO DE OBJETOS

Definir un *scope* es un aspecto básico de cualquier generador de modelos, es la base de los mismos (en caso de que no existiera, no se podría controlar de ninguna manera la forma de los modelos generados). La interpretación de *scope* es similar a la que se da al *scope* de una sesión php: es, por así decirlo, lo que abarca el generador, en qué márgenes se puede mover.

El generador debe ser capaz de recibir un *scope*, en el que se definirán los límites de creación para las distintas clases especificadas (número mínimo y máximo de objetos de cada clase del meta-modelo), y trabajar dentro de esos márgenes.

4.1.2 CARDINALIDAD DE REFERENCIAS

Es interesante también, desde un punto de vista de control sobre los modelos generados, que el usuario pueda especificar otros márgenes distintos a los del meta-modelo para las conexiones del mismo.

El generador debe ofrecer la posibilidad de restringir la cardinalidad de las referencias, siempre y cuando los nuevos márgenes estén contenidos en los márgenes anteriores (los nuevos límites inferior y superior tienen que ser mayor o igual que el antiguo inferior y menor o igual que el antiguo superior, además de que el nuevo superior ha de ser mayor o igual que el nuevo inferior).

4.1.3 CREACIÓN DE REFERENCIAS OPUESTAS

A veces se puede dar el caso de que con la definición del meta-modelo estricta, el usuario no pueda lograr el efecto deseado en los modelos generados. Si nos fijamos en el meta-modelo de la figura 5, la relación "elems" es contenedora, es decir, que si un “Classifier” participa en esa relación, estará contenido por un “Package”. Pero esto no es suficiente si el usuario quiere que todos sus “Classifiers” estén contenidos dentro de un “Package”, y tampoco se podría lograr este efecto cambiando la cardinalidad de la referencia “elems” (no se puede poner el límite inferior a "tanto como “Classifiers” haya").

Debido a esto, el generador tiene que ser capaz de permitir definir referencias opuestas para aquellas que no las tuvieran.

4.2 ESTRATEGIAS DE GENERACIÓN

4.2.1 MODELO MÍNIMO

Una vez que se especifica el espacio de búsqueda, el generador creará la cantidad mínima de objetos indicados para cada clase. Se puede dar el caso que, a la hora de relacionar las clases entre sí, no sea suficiente con los objetos creados para satisfacer una relación y tengamos que recurrir a crear más objetos (siempre sin pasarse del máximo definido para la clase en particular). En este punto, sería interesante que el generador nos permita dos opciones: o bien creamos el número mínimo de objetos necesarios para satisfacer la relación, o bien creamos un número al azar que sea suficiente para completar la relación y no se pase ni del límite superior de la misma ni del máximo definido para la clase en el espacio de búsqueda.

4.2.2 FORMA DE CONECTAR LOS OBJETOS

A la hora de rellenar las conexiones de un objeto, para cada una de las mismas tendremos muchos objetos candidatos. ¿En qué orden elegimos los objetos candidatos para usarlos en la referencia?

El generador debe ofrecer dos posibilidades: o bien elegir los objetos aleatoriamente de entre los objetos candidatos, o bien siguiendo una heurística. A la hora de conectar los objetos, sería interesante, si la referencia tiene una opuesta, elegir antes los objetos que aún no tienen esa referencia opuesta completa (es decir, elegir antes los objetos que aún no han cumplido el mínimo exigido para la referencia opuesta).

El modo de conexión aleatoria no tiene mucha más explicación que el que su nombre indica, simplemente se elegirán los objetos de entre los candidatos al azar.

4.2.3 MODELOS EXTENSIBLES

Como ya se comentó, el generador trabaja con cualquier meta-modelo Ecore existente, o cualquier meta-modelo Ecore válido que cree el usuario. En el segundo caso, probablemente, el meta-modelo sea pequeño y el usuario lo maneje con facilidad y soltura, y tenga claro qué objetos especificar en el espacio de búsqueda si quiere que se puedan generar modelos válidos con esas limitaciones. En cambio, si se está utilizando un meta-modelo ya existente, es muy probable que el usuario no lo conozca en detalle y profundidad, y a la hora de generar modelos, no sea capaz de definir un espacio de búsqueda válido o no tenga sentido definir un *scope* para decenas de clases distintas.

Debido a esto, el generador tiene que permitir un funcionamiento extensible, es decir, debe dar las siguientes opciones: o bien generar modelos cuyos objetos sean exclusivamente los definidos en el espacio de búsqueda, o bien, si fuera preciso, permitir la creación de otros objetos distintos a los definidos para que el modelo generado sea válido.

5. DISEÑO

Una vez que se tienen claras las especificaciones del generador, se plantea el diseño del mismo. Para explicar el diseño seguido, lo dividiremos en dos partes: el diseño global del proyecto y el diseño interno del generador.

5.1 DISEÑO GENERAL

Como ya se ha comentado, el generador realizado ha de ser un plug-in para Eclipse. Además, tiene que ser posible configurar los parámetros de entrada especificados en la sección de análisis (punto 4). Con este planteamiento, se seguirá la siguiente arquitectura:

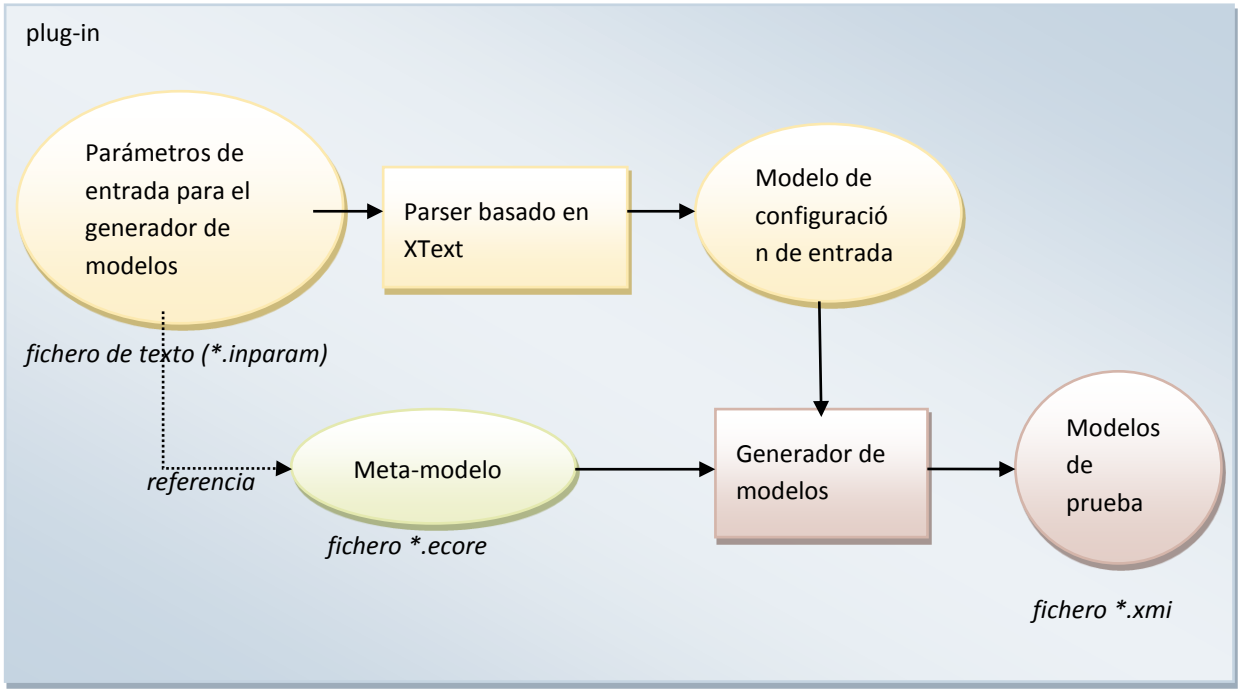

Figura 3. Arquitectura del generador de modelos.

En la figura 3, se muestra el diseño a nivel general de cómo es el proyecto para que el lector se haga una idea de la estructura del mismo. En la práctica, el meta-modelo para el cual se generarán distintos modelos vendrá también indicado en los parámetros de configuración, pero en este apartado se considera importante y más descriptivo mostrarlo por separado para tener una visión global.

Para realizar el *parser*, se ha decidido utilizar XText debido a que su creación es automática si se especifica el formato de entrada para los parámetros en un meta-modelo Ecore (además de que se genera una gramática para el mismo para la que luego, entre otras cosas, dispondremos del menú de auto completar de Eclipse y se mostrará el fichero con formato).

5.2 DISEÑO DEL GENERADOR

El diseño interno del generador es sencillo, puesto que realmente la complejidad del mismo reside en la lógica del algoritmo que logre generar modelos correctos siguiendo las distintas estrategias establecidas. Tiene una clase principal con un método que permite generar modelos a partir de cierta configuración de entrada. Los distintos valores de la configuración se guardarán en atributos de la clase, algunos como tipos primitivos (concretamente, los que especifican cómo debe trabajar el generador) y otros como clases propias (los que especifican el espacio de búsqueda). A continuación se muestra el diagrama UML del diseño para la lógica interna del generador:

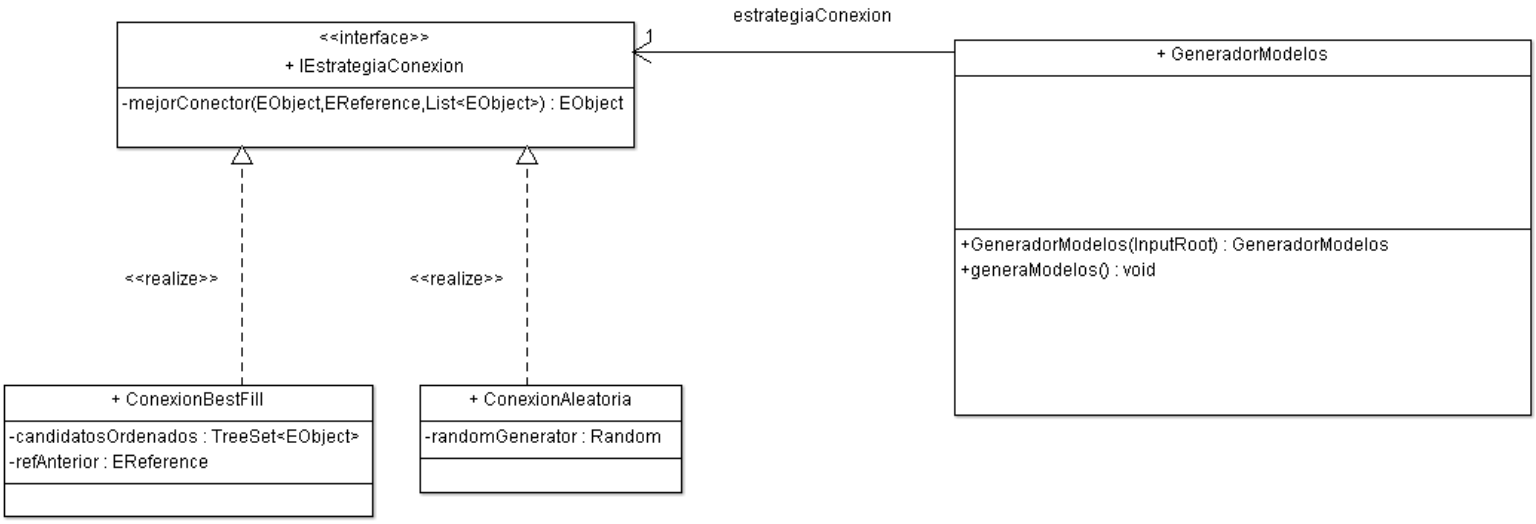

Figura 4. Arquitectura interna del generador.

El diagrama de clases muestra la estructura del generador superficialmente. La clase "GeneradorModelos" tiene bastantes más métodos y atributos que se omiten porque no se consideran necesarios para tener una visión global de la estructura. El espacio de búsqueda (definido en "InputRoot", ver figura 10) llega como entrada al constructor. Una vez se tiene una instancia de esta clase, para generar modelos habría

que invocar al método “generaModelos” (el usuario no tiene que hacer nada y se pone una estructura sencilla).

Cabe destacar del diseño que se ha decidido seguir el patrón "Estrategia" (*Strategy* [19]*)* para especificar la forma en la que se conectan los objetos. Como ya se ha comentado, puede haber muchos criterios distintos (en el proyecto, se han implementado dos estrategias, una que sigue una heurística y otra que conecta aleatoriamente, ver punto 4.2.2), y el patrón "Estrategia" nos permite generalizar los mismos. Se ha pensado hacerlo así para poder extender fácilmente el generador con nuevas estrategias de conexión en un futuro.

6. DESARROLLO

Una vez se tiene claro el análisis del proyecto y se ha planteado el diseño del mismo, se procede al desarrollo, su implementación. Podemos realizar una descomposición del proyecto en tres partes: el plug-in, la gramática para establecer los parámetros de entrada y el generador de modelos en sí.

Siguiendo esta descomposición, el desarrollo del proyecto se ha dividido también en tres fases correspondientes: planteamiento y desarrollo de un algoritmo para generar modelos correctos, creación de una gramática para definir los parámetros de entrada al generador y la configuración de funcionamiento y, finalmente, transformación de todo el proyecto a un plug-in de Eclipse. En los siguientes puntos, contaremos cómo se ha realizado cada fase para la consecución del objetivo final.

6.1 GENERADOR DE MODELOS

La parte del proyecto que consiste en la generación de modelos es la parte principal del mismo.

Como ya hemos comentado en el punto 3.1, el enfoque del generador desarrollado es del tipo *constraint solver*. Recordando brevemente, la idea es que dado cualquier meta-modelo y una gramática genérica, seamos capaces de generar diferentes modelos, añadiendo además posibles restricciones y estrategias de generación para la creación de los mismos.

6.1.1 ¿QUÉ SIGNIFICA CREAR UN MODELO CORRECTO?

Hasta ahora se ha explicado qué es el DSDM, cuál era la motivación del proyecto, qué tecnologías se utilizan en el mismo, qué otros proyectos similares hay y los posibles enfoques a la hora de realizar un generador; pero ¿qué se entiende a la hora de decir "generar un modelo correcto/válido"? Bien, la respuesta es tan sencilla como que, tras crear una serie de objetos impuestos por las restricciones de entrada del usuario, busquemos la forma de conectarlos entre sí satisfaciendo todas las conexiones de los mismos. Vamos a verlo con el siguiente ejemplo:

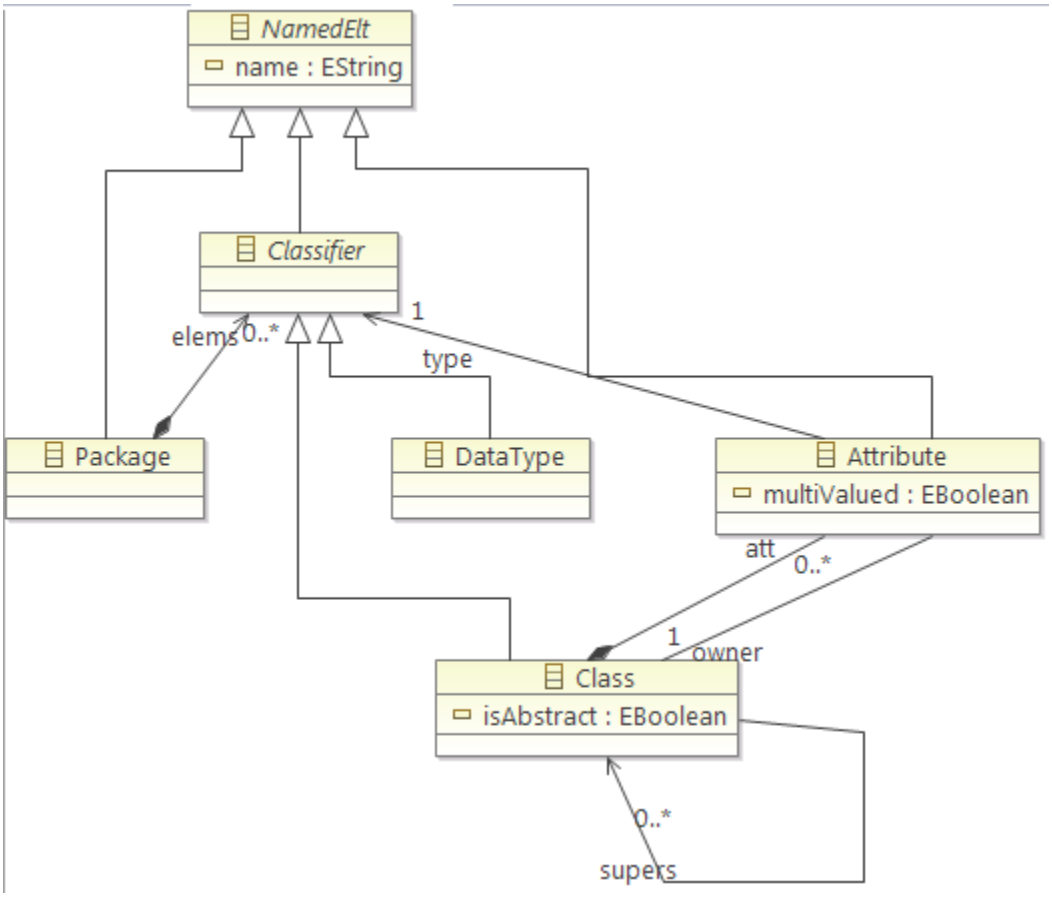

Figura 5. Diagrama de clases del meta-modelo "Class.ecore".

Supongamos que el meta-modelo con el que queremos trabajar es el de la figura 5 y que el espacio de búsqueda que desea el usuario es

```
"Attribute" (1,3)
```

¿Es posible generar un modelo válido para este meta-modelo con esa restricción? La respuesta es no, porque como ya se ha explicado, para que el modelo sea correcto, se han de poder rellenar todas las conexiones de todos los objetos, y para este caso, los atributos necesitan al menos una "Class" y un "Classifier" con qué relacionarse. Si en vez de utilizar el anterior espacio de búsqueda damos el siguiente:

```
"Class" (1,3)
```

Entonces el modelo sería válido simplemente con crear una instancia de "Class", puesto que ni esa meta-clase ni sus meta-clases padres (esto es importante, puesto que para un objeto dado, se tienen que satisfacer todas sus conexiones y las

conexiones que herede) tienen referencias con un límite inferior (o *lowerBound*) mayor que 0. Como último caso introductorio, veamos el siguiente espacio de búsqueda:

```
"Class" (0,2)
"Attribute" (1,1)
```

¿Se podría crear ahora un modelo válido? La respuesta es sí, y el objetivo del generador es que sea capaz de hacerlo (en este caso, partiremos simplemente de un objeto de tipo “Attribute” y sabemos que podemos crear hasta dos objetos de tipo “Class”, aunque basta con uno para poder satisfacer la relación "type" y "owner").

6.1.2 TIPOS DE REFERENCIAS

Como se ha podido ver del meta-modelo anterior, existen diferentes tipos de referencias, y es un factor determinante a la hora de crear modelos correctos. A continuación se analizan los diferentes tipos y en qué afecta cada uno a la hora de la generación:

- **Referencia unidireccional**: Es una referencia a la que a un objeto se le asocian otros, pero estos otros no están asociados con el primero directamente. Es, por ejemplo, como el caso de la referencia "type" del meta-modelo de la figura 5, que va de “Attribute” a “Classifier”. Para rellenar una referencia unidireccional tendremos que no pasarnos del límite superior (o *upperBound*) y, como para cualquier tipo de referencia, satisfacer el *lowerBound* (en definitiva, sólo nos preocupamos por la cardinalidad de la referencia en sí).

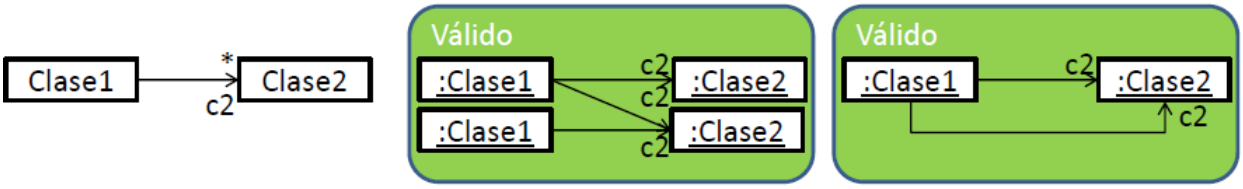

Figura 6. Referencias unidireccionales.

- **Referencia bidireccional**: Es una referencia a la que a un objeto se le asocian otros, y estos otros a su vez están asociados con el primero. Es el caso de las referencias "att" y "owner", ya que una es la opuesta de la otra. A la hora de rellenar una referencia que sí que tiene opuesta establecida, hay que tener en cuenta, además de los límites de la referencia en sí, la cardinalidad de la referencia opuesta.

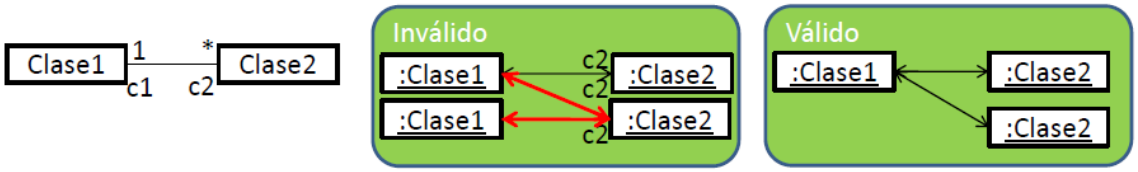

Figura 7. Referencias bidireccionales.

- **Referencia contenedora (relaciones de composición)**: Una referencia, además de ser unidireccional o bidireccional, puede ser también contenedora. En el diagrama del meta-modelo de la figura 5, la referencia "elems", por ejemplo, es contenedora. Si bien a priori no hay una referencia opuesta, y por lo que vimos hasta ahora, no sería inválido que un mismo “Classifier” esté en los "elems" de dos “Package” distintos, un objeto no puede participar en dos conexiones contenedoras al mismo tiempo (es decir, no puede "ser contenido" por dos "padres" a la vez).

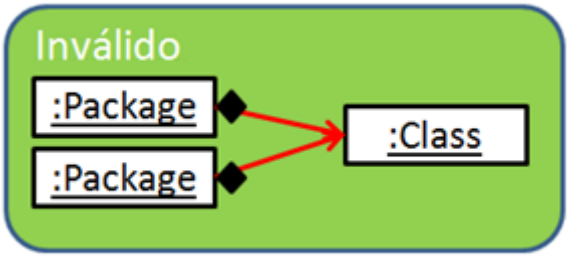

Figura 8. Referencias contenedoras.

6.1.3 MANIPULACIÓN DE MODELOS

Una vez que ya se han visto las bases de qué es lo que se tiene que lograr para obtener un modelo válido, el siguiente paso es desarrollar un algoritmo que satisfaga estas condiciones. Antes de empezar a trabajar en el algoritmo en sí, el primer paso es ser capaces de manejar los modelos: crearlos, cargarlos, guardarlos, añadirles objetos, etc. Para dar solución a este problema, se ha utilizado una librería base ya realizada por Jesús Sánchez (profesor junto con el que se ha trabajado en el proyecto de generación de código C), que, si bien hubo que adaptar un método por la forma de trabajar del generador, nos brindaba ya de por sí todas estas utilidades.

6.1.4 ALGORITMO DE GENERACIÓN

En este punto del proyecto, ya se tiene claro qué es lo que tenemos que hacer y, además, tenemos una librería que nos permite manejar modelos con soltura. El siguiente paso es, entonces, desarrollar la parte del generador que se encarga de que los modelos creados sean correctos.

La primera aproximación del algoritmo sería la siguiente:

1. Crear los objetos mínimos especificados en el espacio de búsqueda.
2. Para cada objeto del modelo:
    a. Comprobar todas las conexiones que salen del mismo. Si alguna de las mismas tiene su límite inferior mayor que 0, buscamos entre las clases especificadas en el espacio de búsqueda si alguna se puede usar para rellenar la referencia. Si existiera:
        i. Vemos si los objetos creados son suficientes. En caso de serlos, completar la referencia.
        ii. Si no son suficientes, intentamos crear más de los mismos, siempre respetando los límites de creación definidos. Si podemos crear los objetos, los creamos, rellenamos la referencia, y nos guardamos los objetos en una lista pendiente a procesar.
        iii. Siempre hay que tener en cuenta si la restricción tiene opuesta, para respetar la cardinalidad de la opuesta también.
        iv. También se ha de tener en cuenta que si la restricción es contenedora, los objetos candidatos no pueden ya estar contenidos en otra restricción contenedora.
        v. Si no hemos podido rellenar la referencia porque o bien no tenemos objetos suficientes ni los podemos crear, o sí que los hay pero no los podemos usar, lanzar una excepción y finalizar la ejecución.

A continuación se muestra un ejemplo de cómo funcionaría este algoritmo:

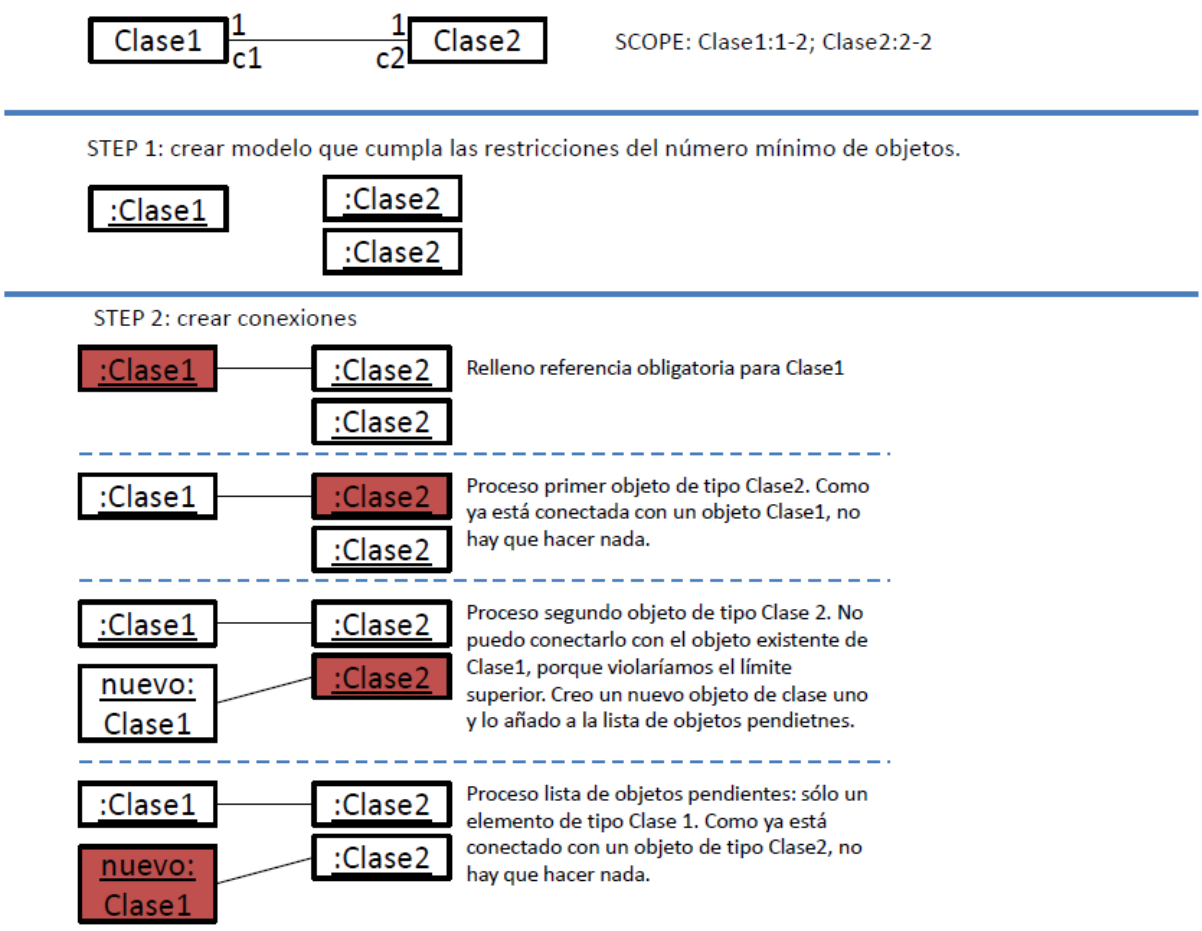

Figura 9. Funcionamiento del algoritmo de generación.

Como primera aproximación está bien, pero a la hora de desarrollarlo, se encuentran varios problemas. Primero y principal, en los meta-modelos Ecore, las referencias *many* se tratan de forma diferente que las diferencias que no lo son (se consideran *many* aquellas que tienen un límite superior mayor que 1). En el caso de que la referencia pueda contener varios objetos, Ecore utiliza una lista para representar a la referencia, mientras que en caso contrario, utiliza simplemente un objeto. Además de esto, la referencia puede o no tener opuesta, puede ser unidireccional o bidireccional, etc.

Para dar una solución inicial a estos problemas, se planteó cada caso posible por separado. Luego, el siguiente paso fue unificar todo y tratar todas las referencias de manera similar: por ejemplo, es lo mismo (a efectos de qué tener en cuenta) que una referencia no tenga opuesta y que la tenga pero no tenga límite superior; también se puede encapsular las diferencias que se plantean entre las conexiones *many* y las que no, para así tratarlas de manera idéntica en el algoritmo principal, etc.

Una vez que se tiene el tratamiento de las referencias unificado, el siguiente problema que nos encontramos es que, en la primera versión, el algoritmo finaliza "si no hemos podido rellenar la referencia porque o bien no tenemos objetos suficientes ni los podemos crear, o sí que los hay pero no los podemos usar". Esta finalización es muy drástica y se puede mejorar: primero, se puede dar el caso de que haya otras clases definidas en el espacio de búsqueda que también se puedan utilizar para rellenar la referencia, la primera encontrada no tiene por qué ser la única; y segundo, si no estamos usando la opción de modelo mínimo (ver punto 4.2.1), puede ser que tengamos objetos "excedentes" en alguna referencia que podríamos quitar de la misma y añadirlos en la que queremos rellenar actualmente. Además, aunque ni con estas dos nuevas condiciones se pudiera rellenar la referencia, si estamos usando la opción de modelo extensible (ver punto 4.2.3), puede que haya otras meta-clases definidas en el meta-modelo y que no están declaradas en el espacio de búsqueda que se puedan instanciar para completar la referencia.

Otra posible especificación de entrada era la posibilidad de añadir referencias opuestas (ver punto 4.1.3). Para implementar esto sin tener que cambiar para nada el cuerpo del algoritmo, simplemente nos basta con añadir dichas referencias al meta-modelo en sí (el algoritmo las tratará igual que trataría a una referencia opuesta real).

Teniendo en cuenta estas consideraciones, la última versión del algoritmo (más detallada) sería:

1. Comprobar que *scope* de cada clase sea válido.
2. Añadir las restricciones opuestas al meta-modelo.
3. Crear los objetos mínimos especificados.
4. Añadir los objetos creados a los objetos pendientes de procesar.
5. Para cada objeto del modelo pendiente de procesar:

5.1. Para cada referencia del objeto:

5.1.1. Mientras que no se satisfaga, recorrer todas las clases definidas en el espacio de búsqueda y obtener los objetos creados.

5.1.2. Mientras la referencia no se satisfaga y la lista de objetos no esté vacía, obtener el siguiente objeto candidato de esa lista según la estrategia de conexión usada (ver punto 4.2.2).

5.1.2.1. Si se puede conectar el objeto (según las restricciones que se han visto antes), conectarlo. Volver al punto 5.1.2

5.1.3. Si se ha rellenado la referencia, volver al punto 5.1. Si, en caso contrario, los objetos de esta clase se han acabado pero la referencia no se ha completado:

5.1.3.1. Intentar usar los objetos existentes de las otras clases definidas. Si rellenamos la referencia con éstos, volver al paso 5.1

5.1.3.2. Si la referencia sigue sin cumplirse, intentar rellenarla añadiendo objetos excedentes de otras referencias (teniendo en cuenta de no violar la cardinalidad de la referencia a la que vayamos a quitar objetos).

5.1.3.3. Si aun así la referencia sigue incompleta, intentar completarla creando objetos de la clase del espacio de búsqueda de la cual hemos obtenido los objetos en el punto 5.1.1 (a la hora de crear objetos, tener en cuenta si estamos usando la estrategia de modelo mínimo o no, ver punto 4.2.1).

5.1.3.4. Se haya o no completado la referencia, volver al punto 5.1.1 para intentar rellenarla creando objetos de otra clase.

5.1.4. Tras recorrer todas las clases definidas en el espacio de búsqueda, si la referencia se ha completado, volver al paso 5.1

5.1.5. Si la referencia no se ha completado, mirar si estamos en modo extensible (ver punto 4.2.3). Si no estamos en modo extensible, lanzar excepción. Si estamos en modo extensible, buscar una clase del meta-modelo que podamos instanciar y no esté en definida en el espacio de búsqueda, rellenar la referencia y volver a paso 5.1. Si no existiera tal clase, lanzar excepción.

6.2 EDITOR DE PARÁMETROS DE ENTRADA

Para la creación del editor de los parámetros de entrada, así como de la correspondiente gramática y su *parser*, se ha decidido utilizar XText, tal y como se explicó en el punto 5.1.

Los parámetros de entrada se guardan como un fichero de texto con extensión "inparam". El editor comprueba automáticamente la corrección del mismo e informa al usuario de posibles errores que pueda contener.

6.2.1 METAMODELO DE LA GRAMÁTICA DE ENTRADA

Para la construcción del proyecto XText, lo haremos a partir de un meta-modelo Ecore que represente la gramática.

Debido al objetivo del meta-modelo que tendremos que crear, el diseño del mismo será sencillo. Crearemos una clase principal que contendrá los parámetros generales del generador y referencias a otros tipos de restricciones o límites del espacio de búsqueda (además se hace así por conveniencia, para crear una gramática poco compleja y que contenga todo lo necesario). El diagrama UML que define el meta-modelo la gramática es el siguiente:

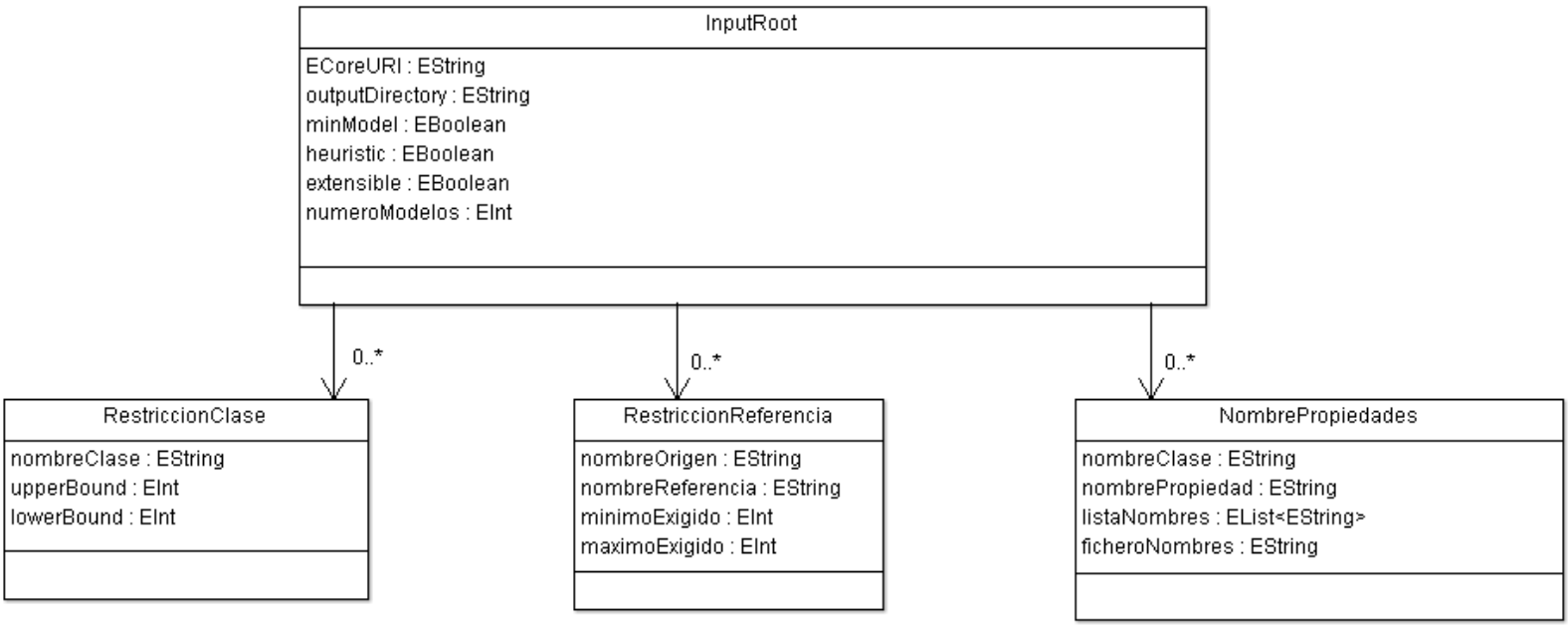

Figura 10. Diagrama UML del meta-modelo de la gramática de entrada.

Una vez se haya definido el fichero Ecore correctamente, crearemos otro de tipo “genmodel” de igual nombre y a partir del mismo. Además de ser utilizado para crear el proyecto XText, el fichero “genmodel” sirve también para generar el código de las clases necesarias para definir los límites de creación de objetos y la forma de conectarlos (las clases de la figura 10). A la hora de obtener la información de un fichero de entrada, la misma será devuelta como instancias de dichas clases, por lo que es altamente recomendable que la lógica del generador también trabaje con las mismas (otra opción sería obtener las instancias de estas clases, procesar la información y crear instancias de otras clases distintas que serían las que se le pasarían como datos a la lógica del generador, pero es más práctico utilizar directamente las que se generan automáticamente).

6.2.2 PROYECTO XTEXT A PARTIR DEL METAMODELO DEFINIDO

El siguiente paso del generador es crear un proyecto XText para definir la gramática en sí y poder disponer de un editor en Eclipse personalizado para la misma. Durante la creación del mismo, definiremos la extensión de los archivos que serán interpretados de acuerdo a la gramática que definamos (en nuestro caso, dicha extensión será "inparam").

Cuando se cree el proyecto, en uno de los ficheros se definirá la gramática. Este fichero se genera con una gramática por defecto, la cual es algo incómoda de utilizar. Para trabajar más cómodamente, se ha cambiado la gramática por defecto y hemos creado una propia (la funcionalidad es la misma, lo único que se ve modificado es la forma en que se definen los parámetros de entrada). La gramática usada es la que se ve en el siguiente recuadro de código:

```
Grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals

import "ParamInput"
import http://www.eclipse.org/emf/2002/Ecore as ecore

InputRoot returns InputRoot:
	(minModel?='minModel')?
	(heuristic?='heuristic')?
	(extensible?='extensible')?
	'ecore' ECoreURI=EString
	('output' outputDirectory=EString)?
	'generate' numeroModelos=EInt
	('with bounds {'
		(listaRestriccionClase+=RestriccionClase ( "," 
listaRestriccionClase+=RestriccionClase)*)?
		(listaRestriccionReferencia+=RestriccionReferencia ( "," 
listaRestriccionReferencia+=RestriccionReferencia)*)?
		(listaOpuestasCreadas+=RestriccionReferenciaInversa ( "," 
listaOpuestasCreadas+=RestriccionReferenciaInversa)*)?
	'}')?
	('withvalues {'
		(listaNombrePropiedades+=NombrePropiedades ( "," 
listaNombrePropiedades+=NombrePropiedades)*)?
	'}')?;


EString returns ecore::EString:
	STRING | ID;

EBoolean returns ecore::EBoolean:
	'true' | 'false';

EInt returns ecore::EInt:
	'-'? INT;

NombrePropiedades returns NombrePropiedades:
	nombreClase=EString'.'nombrePropiedad=EString'from' (('{' listaNombres+=EString
( "," listaNombres+=EString)* '}' ) | (ficheroNombres=EString));

RestriccionClase returns RestriccionClase:
	nombreClase=EString'(' lowerBound=EInt ',' upperBound=EInt ')';


RestriccionReferencia returns RestriccionReferencia:
	nombreOrigen=EString'.'nombreReferencia=EString'(' minimoExigido=EInt ','
maximoExigido=EInt ')';

RestriccionReferenciaInversa returns RestriccionReferencia:
	nombreOrigen=EString'<.'nombreReferencia=EString'(' minimoExigido=EInt ','
maximoExigido=EInt ')';
```

A continuación se muestran dos imágenes de cómo se vería un fichero escrito con la gramática especificada y cómo se integra el menú de autocompletar de Eclipse con la misma:

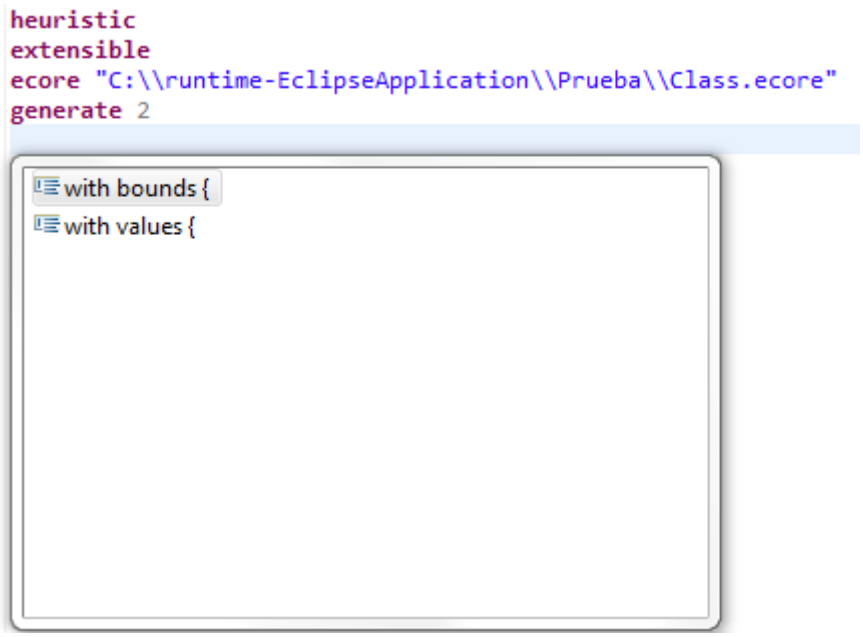

Figura 11. Menú de auto-completar de Eclipse para la gramática diseñada.

```
heuristic
extensible
ecore "C:\\runtime-EclipseApplication\\Prueba\\Class.ecore"
generate 2
with bounds {
    "Class" (2,2)
    "Class"."att" (5,5)
    "Package"<."elems"(1,1)
}
with values {
    "Class"."name" from {"nombre1","nombre2"},
    "Attribute"."name" from "C:/Users/Juan/Desktop/nombreAtrib.txt"
}
```

Figura 12. Ejemplo del editor de Eclipse para la gramática diseñada.

6.2.3 CREACIÓN DE UN FICHERO SEGÚN LA GRAMÁTICA

En este apartado se explicará cómo crear y definir un fichero de configuración de entrada para el generador de modelos, así como qué efecto tiene el poner o no distintos atributos.

Para empezar, tendremos que crear un nuevo fichero con la extensión "inparam". Una vez creado, podremos añadir los siguientes valores, que serán los que modelen el funcionamiento del generador:

- **minModel**: En caso de añadirlo, se especifica al generador que trabaje en modo "minModel", es decir, que genere modelos del menor tamaño posible que cumplan con el espacio de búsqueda definido (ver punto 4.2.1).
- **heuristic**: En caso de añadirlo, se especifica al generador que trabaje en modo "heuristic", es decir, que siga una heurística a la hora de elegir un objeto de los candidatos para rellenar una conexión (ver punto 4.2.2).
- **extensible**: En caso de añadirlo, se especifica al generador que trabaje en modo "extensible", es decir, que se permita crear objetos de clases que no estén definidas en el espacio de búsqueda para lograr que el modelo sea válido (ver punto 4.2.3).
- **ecore**: Este atributo es obligatorio. Se ha de añadir y luego poner una cadena para representar la ruta del fichero Ecore que contiene el meta-modelo para el cual queremos generar modelos.
- **output**: Es un atributo opcional que sirve para indicar la ruta donde se quieren guardar los modelos generados. En caso de no añadirlo, los modelos generados se guardarán en la misma ubicación que se encuentre el fichero “inparam” que se está utilizando.
- **generate**: Este atributo es obligatorio. Indica cuantos modelos se quieren generar.
- **with bounds {}**: Sirve para especificar el espacio de búsqueda con el que queremos que trabaje el generador. En concreto, aquí se pueden definir tres restricciones:

- *Scope* (ver punto 4.1.1): Sirve para especificar qué clases crear. Se crearán clases sólo de las que estén especificadas aquí (salvo que se use el atributo extensible, ver punto 4.2.3) y el número de instancias de la misma se limitará entre el mínimo y máximo especificado entre paréntesis, respectivamente (si el máximo tiene valor negativo, se podrán crear hasta MAX_OBJ instancias de dicha clase, siendo MAX_OBJ una constante interna). Las distintas restricciones de este tipo se han de separar con una "," (coma). Por ejemplo, “Class” (1,2) indicaría que se tiene que generar modelos con entre una y dos clases.
- Restricciones de referencias (ver punto 4.1.2): Sirve para alterar el límite inferior y el límite superior de una referencia del meta-modelo, siempre y cuando los valores especificados no contradigan a los del mismo. La primera cadena indica de qué clase sale la restricción, luego se añade un "." (punto) y luego otra cadena para indicar el nombre de la referencia. Entre paréntesis incluiremos la nueva cardinalidad deseada. Las distintas restricciones de este tipo se han de separar con una "," (coma). Por ejemplo, “Class”.”att” (1,4) indicaría que se tiene que generar modelos en el cual todas las instancias de “Class” tengan entre uno y cuatro “Attributes” en la relación “att”.
- Restricciones opuestas inferidas (ver punto 4.1.3): Sirve para añadir una restricción de cardinalidad en el extremo opuesto de una referencia unidireccional. El formato es igual que en el caso anterior, salvo que hay que añadir un "<" (menor) antes del "." (punto). En este caso, la segunda cadena representa el nombre de la referencia a la que le queremos añadir una opuesta. Por ejemplo, “Package”<.”elems” (1,1) indicaría que para la creación de modelos, se infiera una referencia opuesta a “elems” con cardinalidad (1,1).

- **with values{}**: Sirve para especificar el valor de ciertas propiedades de tipo *String*. La primera cadena representa el nombre de la clase a la cual le queremos dar valor a una de sus propiedades, luego hemos de añadir un "."

(punto), y la segunda cadena representa el nombre de la propiedad en sí. Para dar los valores se puede hacer de dos formas: o bien especificarlos de la forma {"valor1", ... , "valorN"} o bien especificando la ruta a un fichero donde cada posible valor estará en una línea distinta. Si el fichero está vacío, el valor que se le dará a la propiedad, por defecto, será <nombreClase>_<nombrePropiedad>. Por ejemplo, "Class"."name" from {"nombre1","nombre2"} significaría que la propiedad "name" de todas las instancias de "Class" valdría o bien "nombre1" o "nombre2" (también se podría poner "Class"."name" from "ruta de archivo").

6.3 PLUG-IN

Es el final del proyecto. Crearemos un nuevo "Plug-in Project" de Eclipse y sobrescribiremos algunos métodos para lograr el funcionamiento adecuado.

Para la creación del plug-in, usaremos un *template* proporcionado por Eclipse que hace que el mismo sea del tipo pop-up (estará disponible al hacer click derecho sobre un fichero de extensión "inparam", apareciendo como una opción del menú contextual desplegable). Eclipse creará dos clases Java que serán las que definan el comportamiento del plug-in. Concretamente, para establecer qué se quiere realizar una vez que se haya seleccionado la acción del nuevo plug-in creado, tendremos que implementar el código de los métodos:

```
public void selectionChanged(IAction action, ISelection selection) {

}

public void setActivePart(IAction action, IWorkbenchParttargetPart) {

}
```

El primero es un método que se llama automáticamente cada vez que se cambia la selección en el *workbench*. En nuestro caso en concreto, lo único que nos interesa es obtener el fichero sobre el que se ha hecho click (siempre y cuando el fichero sea de tipo “inparam”, en caso contrario, no aparecerá la opción del plug-in en el menú contextual desplegable), por lo que el código del mismo será tan sencillo como:

```
public void selectionChanged(IAction action, ISelection selection) {

        file = (IFile) ((IStructuredSelection) selection).getFirstElement();

}
```

El segundo es el que realiza la acción en sí. No es tan corto como el anterior pero la idea del mismo es igual de simple: ejecutar el generador de modelos con los parámetros de entrada especificados en el fichero "file”. Para ello, tendremos que crear un recurso de EMF a partir del archivo seleccionado y obtener del mismo el primer elemento de su contenido, que será del tipo "InputRoot" (ver punto 6.2). Una vez que lo tengamos, invocaremos al generador, capturaremos su resultado y mostraremos la información pertinente a través de un diálogo.

7. RESULTADOS

En este apartado se mostrará el plug-in en funcionamiento. Para ello, dividiremos la sección en dos apartados: el primero, en el que mostraremos ejemplos realizados sobre el meta-modelo de la figura 5, y el segundo, un ejemplo concreto de uso del generador para el proyecto que da origen al mismo (ver punto 2).

7.1 FUNCIONAMIENTO DEL GENERADOR

7.1.1 LÍMITES INCORRECTOS

En este ejemplo nos centraremos en mostrar el comportamiento del generador cuando damos unos límites inválidos en el *scope* de una clase (si los límites de las referencias no son correctas, se usarán los originales del meta-modelo). Para ello, usaremos el siguiente fichero de configuración, donde lo único importante para este caso es la parte de "with bounds":

```
ecore "C:\\runtime-EclipseApplication\\Pruebas\\Class.ecore"
generate 1
with bounds {
      "Attribute" (2,1)

}
```

El generador detecta el fallo e informa del mismo al usuario:

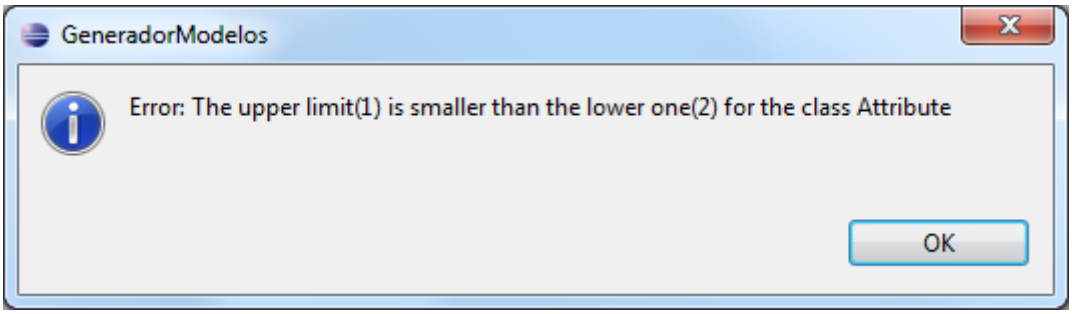

Figura 13. Error por límites incorrectos.

7.1.2 OBJETOS INSUFICIENTES

En este ejemplo nos centraremos en mostrar el comportamiento del generador cuando, con los objetos de cada clase especificados en el espacio de búsqueda, no es posible generar un modelo válido. Para ello, usaremos el siguiente fichero de configuración, donde es importante que no esté el atributo "extensible" (para que los objetos se limiten solamente a los especificados, ver punto 4.2.3) y que no demos información para ninguna clase que herede de “Classifier”, así el “Attribute” generado no tiene ningún objeto que pueda completar su relación "type" y "owner":

```
ecore "C:\\runtime-EclipseApplication\\Pruebas\\Class.ecore"
generate 1
with bounds {
      "Attribute" (1,1)

}
```

El generador detecta el fallo e informa del mismo al usuario:

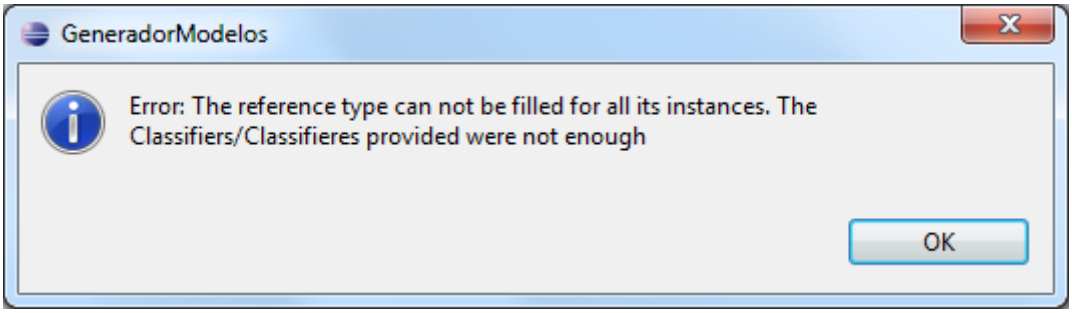

Figura 14. Error por falta de objetos.

El generador informa del error al encontrar la primera referencia que no se puede rellenar, en este caso, "type".

7.1.3 ENTRADAS CORRECTAS

En este punto nos centraremos en mostrar cómo funciona el generador con ficheros de entrada correctos y el/los modelo/s generado/s.

Para el primer caso, probaremos con el mismo fichero que en el caso anterior pero añadiendo el atributo “extensible”.

```
extensible
ecore "C:\\runtime-EclipseApplication\\Pruebas\\Class.ecore"
generate 1
with bounds {
       "Attribute" (1,1)

}
```

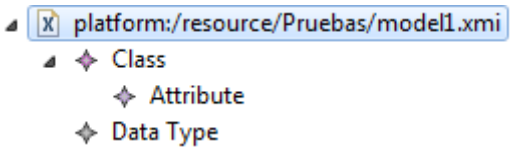

Figura 15. Modelo generado (1).

Como vemos, el generador ha creado un objeto de tipo "Data Type" y otro de tipo "Class" para completar las relaciones "type" y "owner", respectivamente, del “Attribute” creado. Para ver que el "Data Type" está asociado con el "Attribute", como la relación no es contenedora, hay que verlo en las propiedades del mismo:

Property	Value
Multi Valued	false
Name	
Owner	Class
Type	Data Type

Figura 16. Propiedades de un objeto tipo “Attribute”.

Por último, mostraremos un caso de entrada más complejo. Intentaremos crear un modelo con un objeto de tipo "Package" y tres de tipo "Class". Además, diremos que todos los "Classifier" tienen que estar contenidos en un "Package" creando una relación inversa a "elems". También cambiaremos la cardinalidad de la relación "att" de "Class", de (0..*) a [1..4], para forzar que todas las clases tengan entre 1 y 4 atributos, y añadiremos el atributo "extensible" para que así el generador cree los atributos que necesite. Daremos un fichero de entrada con una lista de posibles nombres a los "Attributes", y escribiremos manualmente una lista de posibles nombres para las "Classes" en el fichero de entrada. Con estas especificaciones, tendríamos el siguiente fichero:

```
extensible
ecore "C:\\runtime-EclipseApplication\\Pruebas\\Class.ecore"
generate 5
with bounds {
	"Package" (1,1),
	"Class" (3,3)
	"Class"."att" (1,4)
	"Package"<."elems" (1,1)
}
with values {
	"Class"."name" from {"clase1","clase2","clase3","clase4","clase5"},
	"Attribute"."name" from "C:\\Users\\Juan\\Desktop\\nombreAtrib.txt"

}
```

El resultado que obtenemos es el esperado:

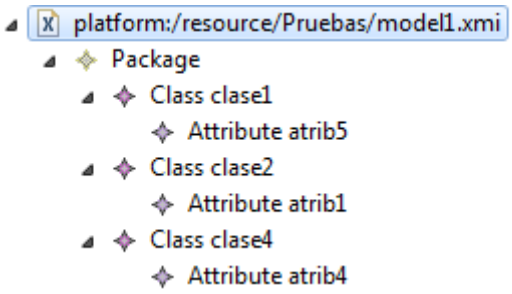

Figura 17. Modelo generado (2).

Si observamos los 5 modelos que se generan (**generate** 5), todos los objetos de tipo "Class" sólo tienen un atributo. Esto es una particularidad de la forma de trabajar "extensible", ya que si el generador tiene que crear objetos no especificados en el espacio de búsqueda, crea los justos y necesarios. Si en vez del fichero anterior, hubiéramos utilizado el siguiente, en el que añadimos la opción “minModel” y especificamos un *scope* para “Attribute” (la opción “extensible” es irrelevante que esté o no en este caso, ya que con las clases especificadas en el espacio de búsqueda se pueden generar modelos válidos), el resultado sería el mismo:

```
minModel
ecore "C:\\runtime-EclipseApplication\\Pruebas\\Class.ecore"
generate 5
with bounds {
	"Package" (1,1),
	"Class" (3,3),
	"Attribute" (0,5)
	"Class"."att" (1,4)
	"Package"<."elems" (1,1)
}
with values {
	"Class"."name" from {"clase1","clase2","clase3","clase4","clase5"},
	"Attribute"."name" from "C:\\Users\\Juan\\Desktop\\nombreAtrib.txt"

}
```

Si el objetivo es lograr que los objetos de tipo "Class" tengan realmente entre 1 y 4 "Attributes", entonces tendremos que especificar que queremos "Attributes" en el espacio de búsqueda y no trabajar en modo "minModel”. En el siguiente ejemplo se mostrará este caso, limitando los "Attributes" entre 0 y 8. Como el algoritmo implementa una función que quita objetos excedentes de relaciones, nunca se podría dar el caso de no poder generar un modelo válido porque el generador cree todos los “Attributes” de los que dispone al rellenar la referencia de las dos primeras “Classes” (distribuyendo 4 para cada una), ya que al llegar a la tercera y ver que no la puede completar, le quitaría un "Attribute" a alguna de las otras dos relaciones. El fichero de configuración para este caso, es, entonces:

```
ecore "C:\\runtime-EclipseApplication\\Pruebas\\Class.ecore"
generate 5
with bounds {
	"Package" (1,1),
	"Class" (3,3),
	"Attribute" (0,8)
	"Class"."att" (1,4)
	"Package"<."elems" (1,1)
}
with values {
	"Class"."name" from {"clase1","clase2","clase3","clase4","clase5"},
	"Attribute"."name" from "C:\\Users\\Juan\\Desktop\\nombreAtrib.txt"

}
```

A continuación se muestran dos salidas distintas (ambas válidas):

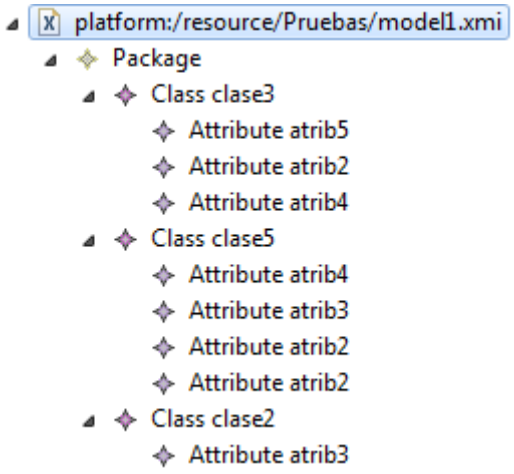

Figura 18. Modelo generado (3).

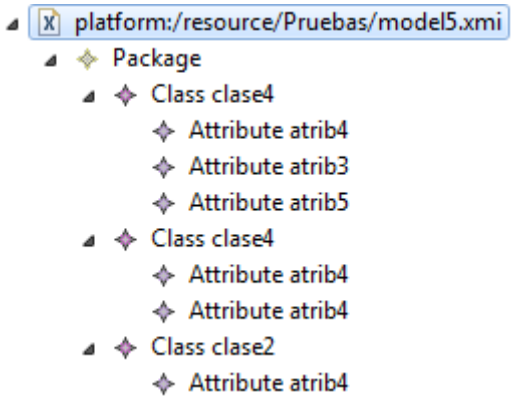

Figura 19. Modelo generado (4).

7.1.4 TIEMPO DE EJECUCIÓN

Hasta ahora, en los ejemplos mostrados, no se ha comentado nada sobre el atributo "heuristic" (ver punto 4.2.2). Esto es porque el uso o no de este atributo no afecta a la forma que tendrá el modelo final, sino que sólo indica si se seguirá una heurística a la hora de conectar objetos o si la conexión se realizará de forma aleatoria. El seguir la heurística propuesta en el punto 4.2.2 podría ahorrar luego al algoritmo tener que buscar objetos sobrantes de ciertas conexiones, y, por tanto, mejorar su tiempo de ejecución. Por otra parte, el proceso de elegir al mejor candidato según la heurística podría ser más costoso que el de obtener objetos de conexiones sobrantes, ya que lo segundo es un caso hipotético, que quizás no se dé, mientras que el aplicar la heurística se haría siempre. En este ejemplo, por tanto, vamos a medir el tiempo de ejecución del programa global, y vamos a comprar los tiempos que se obtienen si se usa "heuristic" o no.

En la primera medición usaremos el siguiente fichero de entrada:

```
ecore "C:\\runtime-EclipseApplication\\Pruebas\\Class.ecore"
generate 100
with bounds {
	"Package" (5,5),
	"Class" (250,250),
	"Attribute" (0,500)
	"Class"."att" (1,4)
	"Package"<."elems" (1,1)
}
with values {
	"Class"."name" from {"clase1","clase2","clase3","clase4","clase5"},
	"Attribute"."name" from "C:\\Users\\Juan\\Desktop\\nombreAtrib.txt"

}
```

Si antes de comprobar los resultados hacemos un análisis, teóricamente el tiempo de ejecución para esta configuración tendría que ser menor que si hubiéramos puesto "heuristic" debido a que no estamos creando "Attributes" inicialmente, sino que sólo se crean a la hora de rellenar la relación "att". Si usamos la configuración "heuristic", el algoritmo perderá tiempo intentando buscar la mejor opción a la hora de rellenar las referencias, y en este caso daría igual. El tiempo empleado para este

fichero, sin esta opción, es en torno a los 4,7 segundos (hay que tener en cuenta que se están generando 100 modelos con muchos objetos):

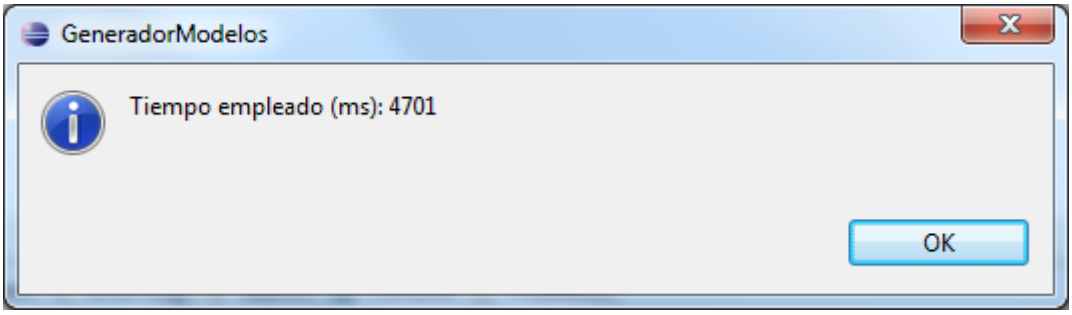

Figura 20. Tiempo de ejecución (1).

Si al mismo fichero de entrada añadimos la opción "heuristic", en cambio, la ejecución ronda los 15 segundos:

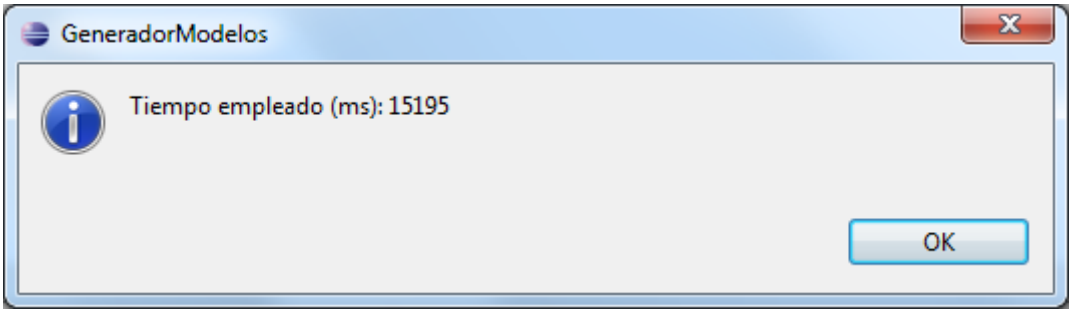

Figura 21. Tiempo de ejecución (2).

Como habíamos anticipado, con una configuración de este tipo, es más rentable conectar los objetos de forma aleatoria.

En la segunda medición vamos a cambiar el fichero de entrada al siguiente:

```
ecore "C:\\runtime-EclipseApplication\\Pruebas\\Class.ecore"
generate 100
with bounds {
	"Package" (5,5),
	"Attribute" (250,250),
	"Class" (250,250)
	"Class"."att" (1,4)
	"Package"<."elems" (1,1)
}
with values {
	"Class"."name" from {"clase1","clase2","clase3","clase4","clase5"},
	"Attribute"."name" from "C:\\Users\\Juan\\Desktop\\nombreAtrib.txt"

}
```

Como se puede ver, ahora estamos dando el número exacto de atributos para que el modelo sea válido y se creen todos al principio. Además, se están creando antes de las "Classes". Este detalle es importante, ya que haciéndolo así, primero se rellenarán las referencias de los "Attributes", y se podría dar que los 250 "Attributes" tengan el mismo "owner", forzando a que el algoritmo se metiera en la función de buscar elementos excedentes de otras referencias al intentar rellenar la relación "att" de las demás "Classes". Si se rellenaran primero las referencias de "Classes", como sólo se rellena hasta satisfacer la relación, a cada clase se le asignaría un "Attribute" libre, y el algoritmo nunca se metería en el método para buscar elementos excedentes de otras referencias. Primero se comprobará que esto es correcto, así que se medirá el tiempo con este fichero y otro igual pero creando las "Classes" antes que los atributos:

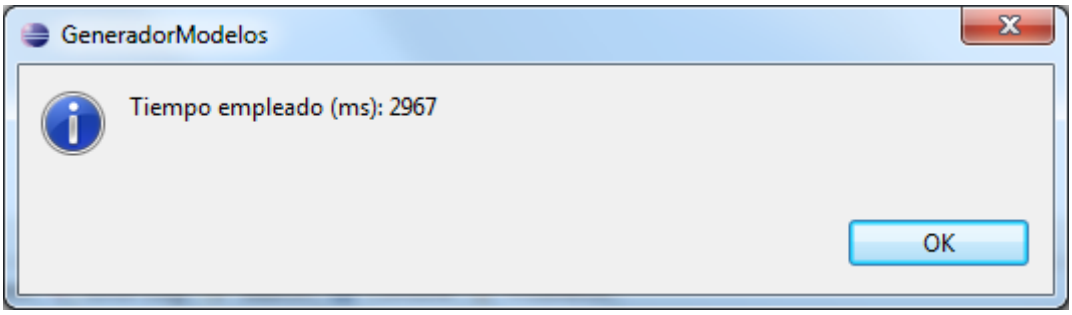

Figura 22. Tiempo de ejecución (3).

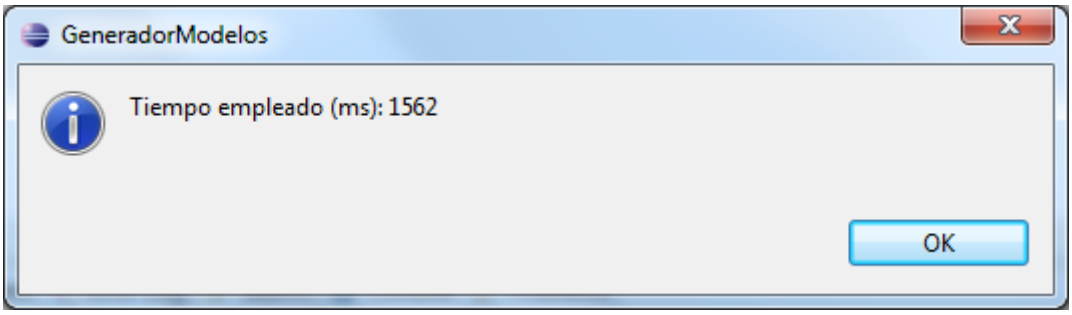

Figura 23. Tiempo de ejecución (4).

Como hemos explicado, en el caso de la figura 23, donde primero se rellenan las referencias de las "Classes" (se ponen antes que los "Attributes" en el espacio de búsqueda), el tiempo es casi la mitad que en de la figura 22, donde primero se están rellenando las referencias de los "Attributes" y el algoritmo se mete en el método de buscar elementos excedentes de otras referencias. A continuación usaremos el siguiente fichero, en el que añadimos la opción "heuristic":

```
ecore "C:\\runtime-EclipseApplication\\Pruebas\\Class.ecore"
generate 100
with bounds {
	"Package" (5,5),
	"Attribute" (250,250),
	"Class" (250,250)
	"Class"."att" (1,4)
	"Package"<."elems" (1,1)
}
with values {
	"Class"."name" from {"clase1","clase2","clase3","clase4","clase5"},
	"Attribute"."name" from "C:\\Users\\Juan\\Desktop\\nombreAtrib.txt"

}
```

Evidentemente, esta configuración tendrá que ser más lenta que la de la figura 23, ya que en ninguno de los dos casos nos tendremos que meter en el método de buscar elementos excedentes, pero en el segundo se pierde tiempo con la heurística. Lo importante de esta prueba es ver si con la heurística, se mejora el tiempo de la figura 22, que es el caso en el que el algoritmo sí se mete en la función anteriormente citada.

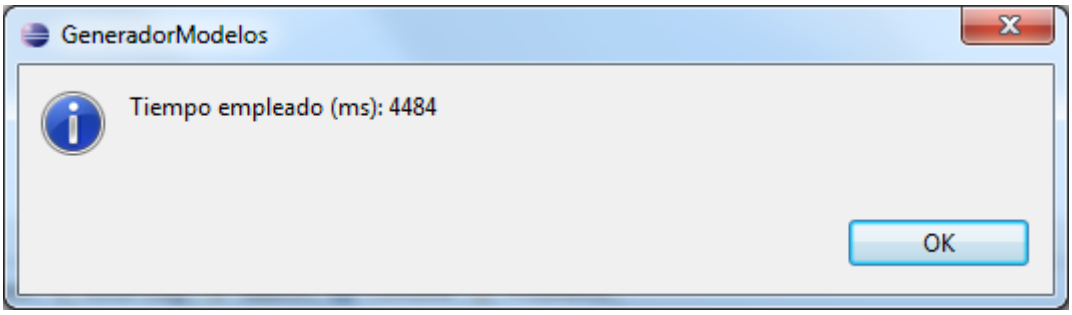

Figura 24. Tiempo de ejecución (5).

Usando la heurística se ha comprobado que el algoritmo, aun creando los "Attributes" antes, no se mete en el método de añadir de excedentes, pero aun así, el tiempo que se pierde con la heurística es mayor que el que se pierde en buscar elementos excedentes de otras conexiones.

Como conclusiones de estas mediciones, es interesante el hecho de que usar la heurística no es rentable hablando en tiempos de ejecución, puesto que la alternativa, buscar elementos excedentes de otras relaciones, es más rápida. Además, es muy importante la comparativa de las figuras 23 y 22, ya que se está usando exactamente la misma configuración de entrada, salvo que en una se rellenan unas referencias antes que en otras, y se está consiguiendo reducir el tiempo de ejecución prácticamente a la mitad. Si el usuario conoce el modo de funcionamiento del algoritmo, puede crear el fichero de entrada de forma conveniente para optimizar el tiempo empleado en generar los modelos.

7.2 GENERACIÓN DE DIAGRAMAS DE TRANSICIÓN DE ESTADOS

Como se ha mencionado en la introducción de esta sección, en este apartado mostraremos cómo se comporta el generador de modelos en un entorno real de trabajo. Para ello, generaremos un modelo de una máquina de estado y luego usaremos ese modelo como entrada del generador de código C descrito en el punto 2. Intentaremos generar la siguiente máquina de estados, la más sencilla que se nos pueda ocurrir:

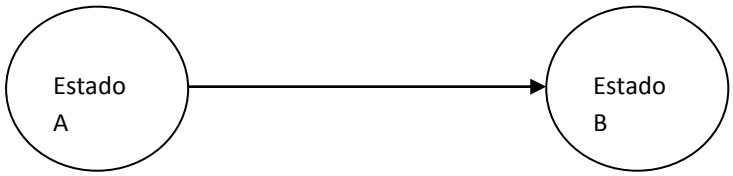

Figura 25. Ejemplo máquina de estados (1).

El problema que nos encontramos a la hora de intentar realizar esto, es que las referencias que conectan estados entre sí son de cardinalidad (0..n), por lo que el generador no las intenta completar. Las únicas conexiones que se han de rellenar son “source” y “target” del objeto de tipo “Transition”, y nada impide que “source” y “target” valgan lo mismo, por lo que se podría dar la siguiente máquina de estados (que sería válida para el generador de código, pero carece de sentido):

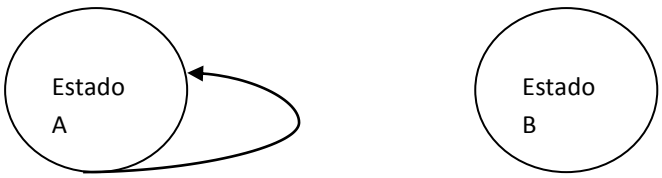

Figura 26. Ejemplo máquina de estados (2).

Con el estado del generador actual, no hay solución posible a este problema. Incluso aunque tuviéramos dos transiciones y forzáramos a que las relaciones "incoming" y "outgoing" de los estados tuvieran cardinalidad mínima 1, se podría dar el mismo caso que el de la figura 26 y que B tenga también otra autotransición (si existiera un mecanismo para evitar que dos propiedades de un objeto valgan distinto, se podría logar, pero las autotransiciones son válidas en las máquinas de estados, así que no tendría sentido "prohibirlas"). Además, nada impide que los dos estados tengan el mismo nombre (tal y como está diseñado ahora, se elige un nombre al azar de entre los nombres proporcionados). Con estas limitaciones, la única solución es generar varios modelos, y seleccionar de los mismos los que nos interesan (en nuestro caso, generaremos varios modelos y elegiremos uno que sea como el de la figura 25).

Una vez que se obtiene el modelo deseado, se puede usar el mismo como entrada del generador de código C sin más problema. Hay muchos aspectos característicos de los modelos de máquinas de estados que se omiten (como por ejemplo, que estos objetos tienen que estar contenidos en una región, la región en una máquina de estados, la máquina tiene que estar asociada a una clase, etc.) ya que son fácilmente controlables gracias a la flexibilidad de la que se dispone al describir el espacio de búsqueda.

8. CONCLUSIONES Y TRABAJO FUTURO

Como ya se ha comentado, debido a las ventajas que ofrece, el DSDM tiene altas probabilidades de ser el nuevo paradigma al que se oriente la programación en un futuro. Además, Eclipse, como herramienta de desarrollo de software, es junto con NetBeans [20] la más utilizada actualmente entre los desarrolladores de Java (según diversas fuentes, se estima que entre el 2011 y el 2012, el 65-70% de los desarrolladores de Java usan Eclipse como IDE principal). Debido a esto, es una gran ventaja de cara al futuro disponer de un plug-in para Eclipse que facilite las pruebas de software de programas que usen modelos como dato de entrada. Además, es mucho mejor si este programa es flexible y ofrece la posibilidad de controlar, de alguna forma, las diferentes estrategias a la hora de generar los modelos.

Respecto al generador en sí, las posibles vías de desarrollo futuro son muchas. Así como inicialmente se ha definido los requisitos especificados en el punto 4, se podrían pensar en muchos otros (por ejemplo, que en vez de crear el número mínimo de objetos especificados por el *scope* de las clases, haya una opción de configuración que permita cambiar este comportamiento a otro que genere un número aleatorio definido entre el mínimo y el máximo especificados). Además, debido al uso del patrón *Strategy* (ver punto 5.2), se podrían especificar fácilmente nuevas estrategias de conexión. También, como ya se comentó en el punto 3.1, se podría añadir la posibilidad de dar modelos semilla (es decir, un modelo base del cual partir y agregarle al mismo los objetos especificados en el espacio de búsqueda con las otras limitaciones que se especifiquen).

En general, y para finalizar, los requisitos del generador realizado son los que se han planteado al inicio del mismo y los que se ha pensado que podrían ser interesantes. A la hora de usarlo en entornos reales, se puede dar el caso de que surjan nuevas ideas que estaría bien tenerlas incorporadas (como ha pasado a la hora de usar el generador en el entorno a partir del cual se ha creado, ver punto 7.2). Es por esto que la funcionalidad del mismo se podría estar extendiendo continuamente según se vaya utilizando y vayan surgiendo nuevos requisitos.

GLOSARIO

DSL (*Domain Specific Language*)

Un DSL (lenguaje específico de dominio) es un lenguaje de programación o especificación de un lenguaje dedicado a resolver un problema en particular, representar un problema específico y proveer una técnica para solucionar una situación particular.

Framework

Es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que puede servir de base para la organización y desarrollo de software. Típicamente, puede incluir soporte de programas, bibliotecas, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto.

IDE (*Integrated Development Environment*)

Un IDE (entorno de desarrollo integrado) es un programa informático compuesto por un conjunto de herramientas de programación (editor de código, compilador, depurador, interfaz gráfica, etc.) para uno o varios lenguajes de programación.

JDT (*Java Development Tools*)

Proyecto de Eclipse que provee un IDE para Java.

Metamodelo

Tipo especial de modelo que especifica la sintaxis abstracta de un lenguaje de modelado. Define la estructura, semántica y restricciones de una familia de modelos.

Modelo

Representación simplificada de una realidad descrito de acuerdo a un meta-modelo específico.

Plug-in

Componente de software que añade una o varias funcionalidades específicas a una aplicación de software existente.

UML (*Unified ModellingLanguage*)

Lenguaje de modelado de propósito general de sistemas de software. Es el más conocido y extendido en la actualidad.

Workbench

Es un componente de la arquitectura de Eclipse. Engloba las distintas vistas, editores, perspectivas y asistentes.

XMI (*XML Metadata Interchange*)

Es un estándar del OMG [14] para intercambiar meta-información a través de XML.

XML (Extensible *Markup Language*)

Es un lenguaje de marcas desarrollado por el W3C [21] (*World Wide Web Consortium*) que permite definir la gramática de otros lenguajes específicos.

REFERENCIAS

[1] http://www.uml.org/

[2] http://www.cafsignalling.com/

[3] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A Challenging Model Transformation. In Int. Conf. on Model Driven Engineering Languages and Systems, pages 436-450. Springer, Sept. 2007.

[4] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL models using Boolean satisability. In Design, Automation and Test in Europe, pages 1341-1344. IEEE, Mar. 2010.

[5] J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verication of UML/OCL models using constraint programming. In Int. Conf. on Automated Software Engineering, pages 547-548. ACM, Nov. 2007.

[6] E. Guerra. Specification-driven test generation for model transformations. In ICMT'12, volume 7307 of LNCS, pages 40–55. Springer, 2012

[7] C. A. C. Pérez, F. Buettner, R. Clarisó, and J. Cabot. EMFtoCSP: A Tool for the Lightweight Verication of EMF Models. In Formal Methods in Software Engineering: Rigorous and Agile Approaches, June 2012.

[8] F. Fleurey, B. Baudry, P.-A. Muller, and Y. Traon. Qualifying input test data for model transformations. Software and Systems Modeling, 8:185–203, 2009

[9] S. Sen, B. Baudry, and J. Mottu. Automatic model generation strategies for model transformation testing. In ICMT'09, volume 5563 of LNCS, pages 148–164. Springer, 2009.

[10] M. Gogolla and A. Vallecillo. Tractable model transformation testing. In ECMFA'11, volume 6698 of LNCS, pages 221–235. Springer, 2011

[11] S. Sen, J.-M. Mottu, M. Tisi, and J. Cabot. Using models of partial knowledge to test model transformations. In ICMT'12, volume 7307 of LNCS, pages 24–39. Springer, 2012.

[12] http://wiki.eclipse.org/Acceleo

[13] http://www.omg.org/spec/MOFM2T/1.0/

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

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

[16] D. Steinberg, F. Budinsky, M. Paternostro and E. Merks: EMF Eclipse Modeling Framework

[17] http://www.eclipse.org/modeling/emf/

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

[19] http://en.wikipedia.org/wiki/Strategy_pattern

[20] https://netbeans.org

[21] http://www.w3c.es/