UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO

UN ENTORNO DE PRUEBAS DE MUTACIÓN EN ECLIPSE

Autora: Raquel Mota Rabadán

Tutora: Esther Guerra Sánchez

Septiembre 2014

Resumen

Una técnica para medir la efectividad de un conjunto de casos de prueba y ayudar a mejorarlo son las pruebas de mutación. En las pruebas de este tipo, se insertan fallos en un programa para obtener distintas versiones erróneas del mismo. Al error introducido en el programa se le denomina *mutación* y a las nuevas versiones del programa inicial se les llama *mutantes*, y se utilizan para comprobar si, dado un conjunto de casos de prueba, éste es capaz de detectar los fallos introducidos en cada mutante. En caso de no detectarlos, el desarrollador debe proporcionar nuevos casos de prueba que detecten los mutantes creados, mejorando de este modo la calidad del conjunto de casos de prueba inicial.

El objetivo de este Trabajo de Fin de Grado es construir una herramienta de mutación que genere de forma automática mutantes de un programa escrito en el lenguaje ATL (*Atlas Transformation Language*), para posteriormente poder medir la eficacia del conjunto de casos de prueba diseñado para probar el programa.

ATL es un lenguaje de programación para definir transformaciones de modelos, que son un tipo de programa software cuyos argumentos de entrada y salida son modelos. Se utilizan dentro del paradigma de Desarrollo Dirigido por Modelos, que es un método de desarrollo de software en el que los datos que se manejan son modelos. Las mutaciones que se generan y describen en el presente trabajo están orientadas a este tipo de software.

Palabras clave: Desarrollo Dirigido por Modelos, Transformación de Modelos, Pruebas de Mutación, Calidad del Software.

Abstract

Mutation testing is a technique to measure the efficacy of a set of test cases and help to improve it. On this kind of testing, faults are injected into a program to get faulty versions of it. Errors introduced in the program are called *mutations* and the new versions of the original program are called *mutants*, and they are used to check whether, given a set of test cases, this is able to detect the faults introduced in each mutant. If they are not detected, the developer must provide new test cases that detect the created mutants, thus improving the overall quality of the initial test set.

The objective of this Bachelor's Project is to build a mutation tool to generate automatically mutants of a program written with the ATL language (*Atlas Transformation Language*), and measure the efficacy of a set of test cases designed for testing the program.

ATL is a programming language to implement model transformations, which are programs whose input and output arguments are models. They are used in the context of Model Driven Development, which is a software development method where the manipulated data are models. The mutations generated and described on this work are oriented to this kind of software.

Key words: Model Driven Development, Model transformation, Mutation Analysis, Software quality.

Agradecimientos

A mis amigos de siempre, porque gracias a ellos he podido desconectar en muchos momentos y conseguir que los días de estudio fueran más llevaderos.
A mis compañeros de la universidad, porque entre todos hemos sabido sacar lo mejor de los largos días de trabajo en la facultad.
A mis profesores, por todo lo que he podido aprender gracias a ellos.
A mi tutora Esther, gracias por tu paciencia y por toda la dedicación que has puesto para ayudarme a llevar adelante este trabajo de fin de grado.
Y por último y en especial, a mis padres y a mi hermano, porque gracias a ellos he conseguido llegar hasta aquí. Y porque siempre están en los buenos y sobretodo en los malos momentos.

Gracias.

Índice de contenido

1. Introducción **1**
- **1.1 Motivación** **2**
- **1.2 Objetivos** **2**
- **1.3 Estructura de la memoria** **3**

2. Estado del arte **4**
- **2.1 Pruebas de software en la actualidad** **4**
- **2.2 Pruebas para la transformación de modelos** **5**
  - 2.2.1 Generación del conjunto de casos de prueba 6
  - 2.2.2 Función oráculo 7
- **2.3 Tecnologías utilizadas** **7**
  - 2.3.1 Eclipse 7
  - 2.3.2 EMF 8
  - 2.3.3 ATL 8
  - 2.3.4 EMF Compare 9

3. Preliminares **9**
- **3.1 Transformación de modelos** **10**
- **3.2 Descripción del lenguaje ATL** **13**
- **3.3 Definición de las pruebas de mutación** **14**

4. Análisis **17**

5. Diseño **20**

6. Implementación **23**
- **6.1 Adaptación de las pruebas de mutación a transformaciones de modelos descritas en ATL** **23**
  - 6.1.1 Generación de una mutación ATL 24
  - 6.1.2 Generación del porcentaje de mutación 26
- **6.2 Tipos de mutaciones** **27**

7 Caso de estudio **39**
- **7.1 Generación de mutantes** **41**
- **7.2 Calcular porcentajes de mutación** **48**

8 Conclusiones y trabajo futuro **53**

9 Referencias **54**

Índice de figuras

Figura 1. Ciclo de vida del Software 1
Figura 2. Esquema de funcionamiento de una transformación de modelos 10
Figura 3. Metamodelo Families 11
Figura 4. Metamodelo Persons 11
Figura 5. Vista de una transformación ATL 14
Figura 6. Pruebas de mutación 15
Figura 7. Maqueta de la pantalla principal 18
Figura 8. Maqueta del fichero csv generado 19
Figura 9. Diagrama de flujo de la aplicación 20
Figura 10. Diagrama de clases de la aplicación 21
Figura 11. Contenido del fichero xmi con la transformación 24
Figura 12. Ejemplo de metamodelo 28
Figura 13. Metamodelo Grafcet 32
Figura 14. Metamodelo KM3 34
Figura 15. Metamodelo Class 39
Figura 16. Metamodelo Relational 40
Figura 17. Estructura del proyecto Class2Relational 40
Figura 18. Pantalla de inicio de la aplicación 41
Figura 19. Pantalla de selección de archivos de entrada 42
Figura 20. Insertar ruta de los archivos 43
Figura 21. Pantalla de selección de directorio 43
Figura 22. Pantalla de generación de mutantes 44
Figura 23. Pantalla de verificación 45
Figura 24. Estado del proyecto Class2Relational 46
Figura 25. Pantalla de cálculo de porcentajes 49
Figura 26. Ficheros del proyecto Class2Relational 50
Figura 27. Vista del fichero csv de salida para modelos de entrada 51
Figura 28. Pantalla de cálculo de porcentajes aleatorios 52
Figura 29. Vista del fichero csv de salida para modelos aleatorios 52

Glosario

- **Desarrollo Dirigido por Modelos:** método de desarrollo que se basa en la construcción de modelos conceptuales que describen el sistema a desarrollar.

- **Transformación de modelos:** programa que realiza la transformación de un modelo de entrada en un modelo de salida distinto.

- **ATL:** Atlas Transformation Language, es un lenguaje que se usa para implementar los programas de transformación de modelos.

- **Pruebas de mutación:** se utilizan para evaluar la calidad de un conjunto de casos de prueba.

- **Mutante:** versión del programa original con un único error inyectado.

- **Metamodelo:** describe los conceptos de un lenguaje de modelado así como las relaciones entre ellos y las reglas estructurales por las que se rigen.

1. Introducción

En las últimas décadas, las redes informáticas, así como las soluciones software para el usuario, han cobrado una importancia notable. Los usuarios de estos servicios de información, que en los inicios de los mismos les parecía algo lejano e incomprensible, ahora tienen un mayor conocimiento y son más exigentes con la funcionalidad de estos servicios.

Por este motivo, se hace totalmente necesaria la fase de control de calidad dentro del ciclo de vida de cualquier producto o solución informática. Como podemos ver en la Figura 1, las pruebas son una más de las fases del ciclo de vida por las que pasa cualquier producto software.

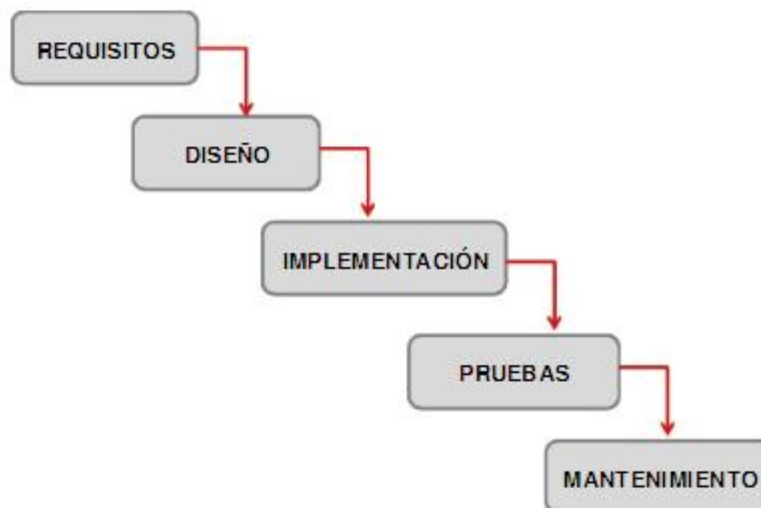

Figura 1. Ciclo de vida del Software

Actualmente, en las empresas de desarrollo informático, cada vez tienen mayor peso los responsables del departamento de control de calidad, que buscan técnicas lo más eficientes posibles para llevar a cabo esta labor. Por este motivo, surge la necesidad de encontrar una buena técnica para detectar fallos en los productos de software, mejorando así la calidad de las soluciones y

proporcionando a los usuarios un servicio más fiable y conforme a sus necesidades.

1.1 Motivación

El Desarrollo Dirigido por Modelos **MDD** (Model-Driven Development) es un método de desarrollo de software que tiene a los modelos como elemento principal en el proceso de desarrollo.

Su objetivo es cerrar la brecha existente entre el diseño de una aplicación y la implementación de la misma. Para ello, los desarrolladores trabajan a un mayor nivel de abstracción (los modelos) usando conceptos propios del dominio de su aplicación. Posteriormente, a partir de esos modelos, es posible automatizar la generación de código para la aplicación final. Este tipo de desarrollo se basa en el uso de modelos y de transformaciones entre modelos. Las transformaciones pueden usarse tanto para traducir entre lenguajes de modelado, como para hacer refactorings y generar código desde los modelos, entre otras funcionalidades. Dado el papel destacado que las transformaciones de modelos juegan en el MDD, se debe garantizar que las transformaciones son correctas. Para ello se deben realizar pruebas, teniendo la certeza de que se han hecho el número suficiente de pruebas. Para esto sirven las pruebas de mutación; para comprobar si un conjunto de casos de prueba es suficientemente completo o no. Actualmente no hay ningún framework open-source desde el que se puedan realizar este tipo de pruebas de mutación para transformaciones escritas en lenguaje ATL [12], que es uno de los lenguajes de transformación de modelos mas usado en la práctica, y esto no resulta útil. Es necesario garantizar que las soluciones que se construyen con este tipo de desarrollo son correctas.

1.2 Objetivos

El objetivo de este Trabajo Fin de Grado es proporcionar una herramienta para medir la **calidad** de un conjunto de pruebas de una transformación ATL. Para

ello, se utilizará una técnica basada en *pruebas de mutación* para, dado un conjunto de modelos de prueba (*test data set*) de una transformación, tener una medida de la calidad de dicho conjunto. Si la calidad es insuficiente, significa que las pruebas realizadas son probablemente insuficientes y se necesite mejorarlas.

En las pruebas de mutación, a una transformación inicial y, aparentemente funcional, se le introducen pequeñas modificaciones, creando distintas versiones erróneas de esa transformación. Estas versiones, son llamadas ***mutantes***, ya que cada una de ellas es una mutación de la transformación original. Cuando se ejecutan estas mutaciones con distintos modelos de entrada (casos de prueba), el resultado debería ser erróneo; es decir, los modelos de prueba deben detectar el error introducido en la transformación original. Si no se detectan errores, se debería mejorar el conjunto de casos de prueba, ya que no es capaz de detectar los errores que hemos creado de manera artificial, y por tanto la probabilidad de que tampoco detecte errores reales en la transformación es mayor.

En este caso, las transformaciones están descritas mediante el lenguaje de transformación ATL (Atlas Transformation Language).

1.3 Estructura de la memoria

El documento consta de seis partes principales donde se explicará en detalle el trabajo realizado: en la sección 2 se explicará el **estado del arte** y las tecnologías. En la sección 3 se dará una explicación general de las transformaciones ATL y de las pruebas de mutación a modo de **preliminares**.
En la sección 4 se explicará a fondo el **análisis** de la aplicación implementada; en la sección 5 se describirá el **diseño** completo de la aplicación y en la sección 6 cómo se ha llevado a cabo la **implementación** de la misma. Posteriormente en la sección 7 se detallarán los **casos de estudio** que se han realizado con los respectivos comentarios acerca de los **resultados obtenidos**.

Por ultimo, el documento finalizará con las **conclusiones** y posibles líneas de trabajo futuro.

2. Estado del arte

2.1 Pruebas de software en la actualidad

El proceso de pruebas de un software [1] es un proceso fundamental en el ciclo de vida de cualquier producto informático, como hemos visto en la sección 1. Este proceso de pruebas consiste en una investigación llevada a cabo con la finalidad de proporcionar información acerca de la calidad de un producto o servicio.

Para llevar a cabo el proceso de pruebas, existen distintos métodos, pero los más utilizados son las pruebas de caja blanca y las pruebas de caja negra. Las pruebas de caja blanca se utilizan para comprobar el funcionamiento interno del software es decir, se diseñan casos de prueba exhaustivos de manera que se ejerzan caminos distintos de ejecución a través del código y se determine si las salidas del programa son apropiadas. Una de las técnicas utilizadas para llevar a cabo las pruebas de caja blanca son las **pruebas de mutación**, en las que se evalúa si el conjunto de pruebas diseñado encuentra todos los posibles errores inyectados en el programa. Por otro lado, en las pruebas de caja negra se trata el software como una caja negra de manera que se comprueba la funcionalidad del programa sin ningún conocimiento de la implementación interna. En este caso, los que realizan las pruebas sólo conocen lo que el software tiene que hacer, pero no tienen idea de cómo lo hace.

2.2 Pruebas para la transformación de modelos

Una vez hemos visto la importancia de la realización de pruebas de software, vamos a aplicarlo ahora al tema de estudio en este trabajo de fin de grado, como son las transformaciones de modelos.

Una transformación coge un modelo de entrada el cual está basado en las especificaciones del metamodelo de entrada, y genera un modelo de salida que está regido por las reglas y definiciones descritas en el metamodelo de salida.
En este caso, puede parecer que para realizar pruebas en este tipo de desarrollo no se plantea mayor reto que el de realizar pruebas sobre el código implementado; pero esto no es cierto, se plantea un reto mucho mayor a la hora de realizar pruebas de este tipo como por ejemplo la selección de un conjunto de casos de prueba que sean relevantes, como sabemos elegir el conjunto de pruebas para llevar a cabo la parte de aseguramiento de calidad de un software no resulta una tarea sencilla, ya que si no se escogen bien los modelos de entrada podemos no estar probando todos los casos necesarios para ver todas las posibles salidas o comportamiento de un determinado programa. Por otro lado, otro de los grandes retos es el de definir una función oráculo, entendiendo como tal una función que se encarga de comprobar todas las salidas del programa y tratar estos resultados para obtener una estimación de la funcionalidad del programa. Vamos a ver mas en concreto cada uno de estos dos retos.

Cabe señalar que al igual que para las pruebas de software, para las pruebas de transformación de modelos se definen también las pruebas de caja blanca[3, 4] y las pruebas de caja negra[5, 6, 7].

2.2.1 Generación del conjunto de casos de prueba

Como se ha mencionado en el párrafo anterior, para realizar pruebas sobre un modelo de transformación, dichas pruebas deben proporcionar modelos de entrada que hayan sido descritos según las especificaciones del metamodelo de entrada. Estos modelos de entrada formarán el conjunto de casos de prueba de una determinada transformación a probar. De manera intuitiva, podemos decir que un conjunto de casos de prueba está completo siempre que todas las clases definidas en el metamodelo fuente sean instanciadas al menos una vez en algún modelo de entrada dentro del conjunto de casos de prueba.

La generación de estos modelos de entrada se puede realizar de dos formas[2]:

- De forma automática: es decir, implementado un programa principal que estudie las características del metamodelo de entrada y a partir de esto genere automáticamente los modelos de entrada que se le pasarían a la transformación para realizar las pruebas.
  Aun que a primera vista, este método parece ser lo mas efectivo, esto no es así, debido a diversos motivos; uno es el elevado número de estrategias que se deben generar para construir estos modelos de entrada, el otro motivo es que los modelos generados automáticamente pueden llegar a ser muy difíciles de entender para la persona que se dedica a realizar estas pruebas de validación, y esto tiene sus consecuencias, como por ejemplo la dificultad que encontrará esta persona al establecer la salida esperada de la transformación.
- Enfoque interactivo: en este tipo de enfoque la persona encargada de este tipo de pruebas proporciona el conjunto de modelos de entrada. Si además se genera un programa que, de forma automática verifique que los conjuntos de prueba creados satisfacen todos los requisitos del modelo de entrada, este tipo de enfoque lo hace mucho mas eficiente que el de generación automática de modelos de entrada.

2.2.2 Función oráculo

Definir este tipo de función resulta fundamental debido a que será ésta la que realice las comparaciones oportunas para determinar si la salida de una transformación es correcta; es decir, la función oráculo es, simplemente, una función que dada la salida de un programa (en nuestro caso ese programa es una transformación) dice si es correcta o no.

En nuestro caso, como se ha usado la herramienta de Eclipse para la generación del proyecto, se usa Eclipse Model Framework Compare (EMFCompare) cuyo entorno permite llevar a cabo las comprobaciones que realiza la función oráculo.

2.3 Tecnologías utilizadas

Se describe a continuación todas las tecnologías que se han usado durante todo el desarrollo de este proyecto, muchas de las cuales han sido novedosas y cuanto menos interesantes.

2.3.1 Eclipse

Eclipse[13] es un entorno de desarrollo integrado (IDE). Contiene un área de trabajo (workspace) y un sistema de plugin ampliable para personalizar el entorno. Está escrito principalmente en Java, y se utiliza sobretodo para desarrollar aplicaciones. Gracias al uso de distintos plugins, Eclipse también puede utilizarse para desarrollar aplicaciones en diversos lenguajes de programación.

Para cada lenguaje de programación utilizado, se reconoce la sintaxis para sombrear las palabras reservadas del lenguaje que se este utilizando.

Además, es muy útil ya que según se va escribiendo el código te va indicando los errores de compilación que estás cometiendo y en muchas ocasiones te proporciona una lista de posibles soluciones a ese error.

2.3.2 EMF

Eclipse Modeling Framework (EMF) [14] es un framework de modelado que aporta la utilidad de generación de código para construir herramientas y otras aplicaciones basadas en un modelo estructurado de datos. Partiendo de la especificación de un modelo (metamodelo) descrito en un fichero .xmi, EMF proporciona las herramientas necesarias para generar un conjunto de clases Java para el modelo (con la especificación del fichero .xmi). Este framework ha resultado ser de gran utilidad debido a que ATL está a su vez definido sobre un metamodelo EMF, y los modelos de entrada y salida que es capaz de gestionar una transformación ATL son también EMF.

2.3.3 ATL

Para visualizar y gestionar los metamodelos y las transformaciones escritas en lenguaje ATL [12], se ha instalado un plugin dentro de eclipse para integrar los proyectos con las transformaciones en este lenguaje junto con el proyecto en código Java para la implementación de nuestra aplicación principal. ATL (Atlas Transformation Language) es un lenguaje de transformación de modelos con distintas herramientas. En el método de desarrollo MDD, ATL proporciona distintas formas para producir un conjunto de modelos de destino a partir de un conjunto de modelos de origen. Está desarrollado sobre la plataforma Eclipse y, su entorno integrado (IDE) proporciona diversas herramientas de desarrollo (resaltado de

sintaxis, depuración, etc.) que tiene como objetivo facilitar el desarrollo de las transformaciones ATL.

2.3.4 EMF Compare

EMF Compare[15] es un plugin de Eclipse que proporciona herramientas para comparar cualquier tipo de modelo EMF. Se puede utilizar de forma manual, de manera que mediante una interfaz de doble ventana podemos ir viendo las diferencias que hay entre dos modelos e ir realizando los cambios que se consideren; también se puede utilizar desde código mediante funciones específicas implementadas y disponibles en este plugin, mediante las cuales en este proyecto se han podido realizar comparaciones entre dos modelos de salida, obteniendo mediante estos métodos el número de diferencias encontradas entre ambos modelos.

Esto ha resultado ser de mucha utilidad, ya que las pruebas de mutación que se han realizado en este proyecto necesitaban comprobar si el modelo de salida generado por una transformación mutante para un modelo de entrada dado, era igual al modelo generado por la transformación original para el mismo modelo de entrada.

3. Preliminares

En esta sección se van a introducir algunos conceptos básicos sobre transformación de modelos y pruebas de mutación, necesarios para entender el desarrollo que se presenta en las siguientes secciones.

3.1 Transformación de modelos

A continuación, la Figura 2 muestra el esquema principal que sigue una transformación de modelos:

Figura 2. Esquema de funcionamiento de una transformación de modelos

En este tipo de programas se manejan distintos términos:

- **Metamodelo**: describe los conceptos de un lenguaje de modelado, las relaciones entre ellos y las reglas estructurales que restringen elementos y combinaciones del modelo con el objetivo de respetar las reglas de dominio. Las figuras 3 y 4 muestran dos ejemplos de metamodelos:

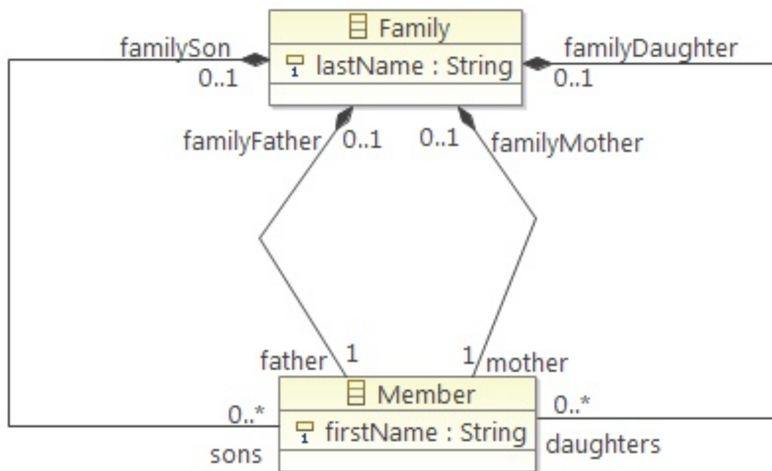

Figura 3. Metamodelo Families

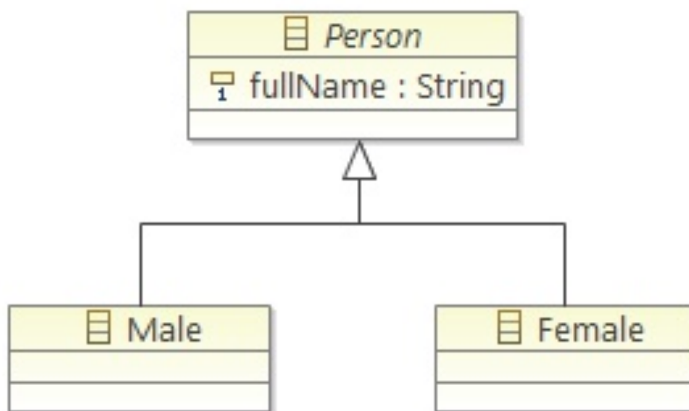

Figura 4. Metamodelo Persons

Como se puede observar, en la Figura 3 se muestra un metamodelo que describe, los objetos de tipo `Family`, y los objetos de tipo `Member` y las relaciones que existen entre ambos objetos. Es decir, en este caso una Familia tiene un apellido a la cual pertenecen distintos miembros: un padre, una madre, y un número arbitrario de hijos e hijas, cada uno de ellos con un nombre de pila.

El metamodelo de la Figura 4 describe los objetos de tipo `Person`, definido como una clase abstracta de la que heredan las clases `Male` y `Female`. Es decir, una persona (ya sea de género masculino o femenino) tiene un nombre completo que se compone de nombre de pila y apellido.

Vemos que estos metamodelos son una representación de la realidad; en el primer caso representa una familia y en el segundo caso representa a una persona en particular. Además, nos especifican también las reglas a tener en cuenta; por ejemplo una familia no puede tener dos madres, pero sí tiene que tener como mínimo una. También podemos ver que se contempla el caso en que una familia no tenga ni hijos ni hijas.

- **Modelo**: Siguiendo las definiciones y reglas que especifica el metamodelo, un modelo describe una instancia válida de metamodelo. Por ejemplo:

```
<Family lastName="March">
	<father firstName="Jim"/>
	<mother firstName="Cindy"/>
	<sons firstName="Brandon"/>
	<daughters firstName="Brenda"/>
</Family>
```

Este es un ejemplo de modelo. Según el metamodelo Families, una familia tiene un apellido (`lastName`) y ciertos miembros, cada miembro con un nombre de pila (`firstName`). Este modelo define a una familia cuyo apellido es "March" y que consta de 4 miembros: un padre de nombre Jim, una madre de nombre "Cindy", un hijo de nombre "Brandon" y una hija de nombre "Brenda".

- **Transformación**: define un conjunto de reglas y definiciones que, aplicadas a un modelo de entrada, obtiene un modelo de salida distinto al original. Esto es lo que se describe en la Figura 2 y se explicará en detalle más adelante.

Existen diversos lenguajes para implementar transformaciones, que se clasifican entre imperativos, declarativos o híbridos, pueden estar basados en reglas o no, y pueden ser gráficos o textuales. En la referencia [17] hay una clasificación exhaustiva de los tipos de lenguajes de transformación existentes. En este trabajo se trabajará con ATL, que es un lenguaje declarativo, textual, basado en reglas para la transformación de modelos.

3.2 Descripción del lenguaje ATL

ATL (Atlas Transformation Lenguage) [12] es un lenguaje de transformación de modelos con un conjunto de herramientas desarrolladas y mantenidas por la compañía de desarrollo OBEO y un grupo de investigación en MDD llamado AtlanMod.

Este lenguaje se emplea para definir las posibles acciones que realiza una transformación. En este proyecto, los metamodelos y modelos de prueba estarán escritos en lenguaje EMF [14], pero las transformaciones están escritas en ATL.

Una transformación se compone de reglas que especifican los cambios que se van a llevar a cabo cuando se ejecute la transformación. En la Figura 5 podemos ver una pequeña parte del código ATL de una transformación determinada.

```
1   module Families2Persons;
2
3   --@path Families=/Families2Persons/Families.ecore
4   --@path Persons=/Families2Persons/Persons.ecore
5
6   create OUT:Persons from IN:Families;
7
8   -- it returns whether a family member is a female
9⊖  helper context Families!Member def: isFemale(): Boolean =
10      if not self.familyMother.oclIsUndefined() then
11          true
12      else
13          if not self.familyDaughter.oclIsUndefined() then
14              true
15          else
16              false
17          endif
18      endif;
19
20  -- it returns the family name of a family member
21⊖ helper context Families!Member def: familyName: String =
22      if not self.familyFather.oclIsUndefined() then
23          self.familyFather.lastName
24      else
```

Figura 5. Vista de una transformación ATL

Este es un ejemplo de transformación que, dado un modelo de tipo `Families` genera otro modelo de tipo `Persons`. En la línea 6 se define la entrada y la salida de la transformación; es decir, el metamodelo de entrada es `Families` y el metamodelo de salida es `Persons`. De la línea 9 a la 18 hay un helper que define un método que se puede invocar sobre los objetos de tipo `Member`. Por último, señalamos también las navegaciones; llamamos navegación al acceso a los diferentes atributos de un determinado objeto. En este ejemplo de la Figura 5, en la línea 23 se define una navegación, ya que estamos accediendo al atributo `lastName` del objeto `familyFather`.

3.3 Definición de las pruebas de mutación

En el apartado anterior 2.2, hablábamos de los dos grandes retos en la validación de las pruebas de transformación de modelos. En concreto, el primero de ellos era generar un conjunto de casos de prueba que tenga en cuenta ciertas

especificaciones del metamodelo de entrada; pero además existe la necesidad de cuantificar cómo de bueno es el conjunto de modelos de prueba que hemos generado. Con esta necesidad surge la idea de realizar pruebas de mutación[8].

Vamos a analizar en detalle cómo se desarrolla el proceso de mutación, desde que obtenemos el conjunto de datos de prueba hasta que finalizamos el proceso; que ocurre cuando el conjunto de datos de prueba que creamos al principio del proceso ha sido probado con todos los mutantes que se han generado.

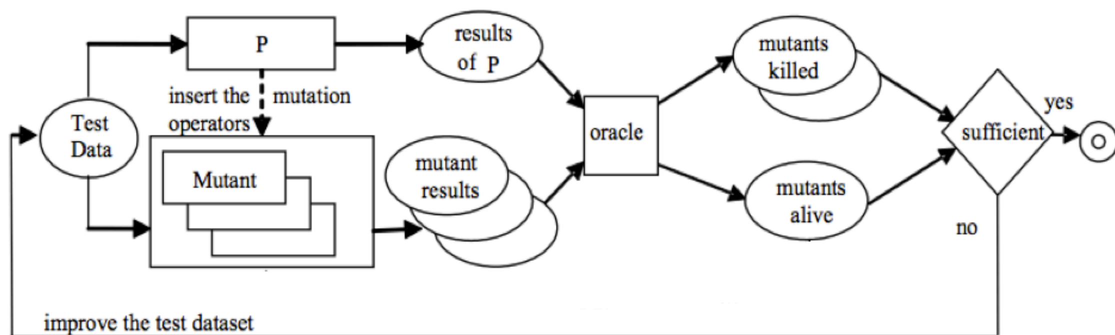

Figura 6. Pruebas de mutación

En la Figura 6, se muestra el proceso completo, que se describe a continuación:

- Dado un conjunto de datos de prueba (Test Data), y un programa original (P), se crean N mutantes de ese programa (la creación de los mutantes se denomina *proceso de mutación*).
- Se ejecuta el programa original con el conjunto de datos de prueba y se obtienen un resultado (results of P).
- Se ejecuta un mutante (Mutant) con el conjunto de datos de prueba y se obtienen uno resultado (Mutant result).
- Se compara el resultado obtenido con el programa original con el resultado obtenido con el mutante y, si los resultados coinciden el mutante sigue “vivo”. Si por el contrario, los resultados no coinciden, este mutante “muere” y es descartado para el resto de casos de prueba.
- Se calcula el porcentaje de mutación mediante la fórmula matemática:

$$\text{Porcentaje de mutación} = \frac{\text{número de mutantes "muertos"}}{\text{número total de mutantes generados}}$$

Si el porcentaje de mutación obtenido se considera suficiente, no se generan mas conjuntos de prueba, y mediante este porcentaje se puede tener una idea fundamentada de cómo de eficaz es nuestro conjunto de datos de prueba.

- Si por el contrario, el porcentaje de mutación obtenido es insuficiente, aún se puede seguir mejorando el conjunto de datos de prueba; se le aplica una mejora y se vuelve a repetir el proceso para obtener nuevos resultados.

El desarrollo de este tipo de pruebas de mutación para la transformación de modelos se propuso por primera vez en [9]. La idea de utilizar este tipo de pruebas en el modelo de desarrollo dirigido por modelos surge porque las técnicas clásicas que se usan para llevar a cabo las pruebas en este tipo de desarrollo no están del todo bien adaptadas a los significativos cambios que este paradigma de software ha inducido en el proceso de desarrollo. Por este motivo, se decide utilizar pruebas de mutación para el desarrollo dirigido por modelos.

Dependiendo del cambio (mutante) que se introduzca en la transformación original se proponen distintos tipos de mutación; relacionadas con la navegación dentro de un metamodelo de entrada (Relation to the Same Class Change, Relation to Another Class Change, Relation Secuence Modification with Addition y Relation Secuence Modification with Deletion), relacionadas con la operación de filtrado que se realiza en la transformación (Collenction Filtering Change with Perturbation, Collection Filtering Change with Deletion y Collection Filtering Change with Addition) y relacionadas con la creación de elementos dentro de la transformación (Class Compatible Creation Replacement, Clasess Association Creation Deletion y Clasess Association Creation Addition).

En este TFG, se han implementado todas las mutaciones de tipo navegación. Las secciones 4, 5 y 6 recogen los detalles de dicha implementación, así como su uso para calcular el porcentaje de mutación de un conjunto de modelos de prueba.

Actualmente existen lenguajes de transformación de modelos para los que se han implementado estos operadores de mutación, como Kermeta[10] o el lenguaje Tefkat[11]. Sin embargo, uno de los lenguajes más usados en la transformación de modelos es ATL[12] el cual se ha usado para este trabajo y para el que actualmente no hay implementación de los distintos tipos de mutación y, dado el gran uso extendido de este lenguaje resulta necesario disponer de una implementación que permita realizar pruebas de mutación.

4. Análisis

La aplicación que se propone consiste en un software para generar algunos de los mutantes posibles de una transformación ATL para, posteriormente calcular el porcentaje de mutantes que han sido detectados según un conjunto de casos de prueba y, medir así la calidad de ese conjunto.

Esta aplicación realizará dos acciones fundamentales; por un lado generarán todos los mutantes de navegación que se puedan crear en una cierta transformación, por otro lado el programa calculará el porcentaje de mutación para, finalmente poder valorar un conjunto de modelos de prueba determinado.

En la Figura 7 vemos la primera maqueta de la pantalla principal de nuestra aplicación:

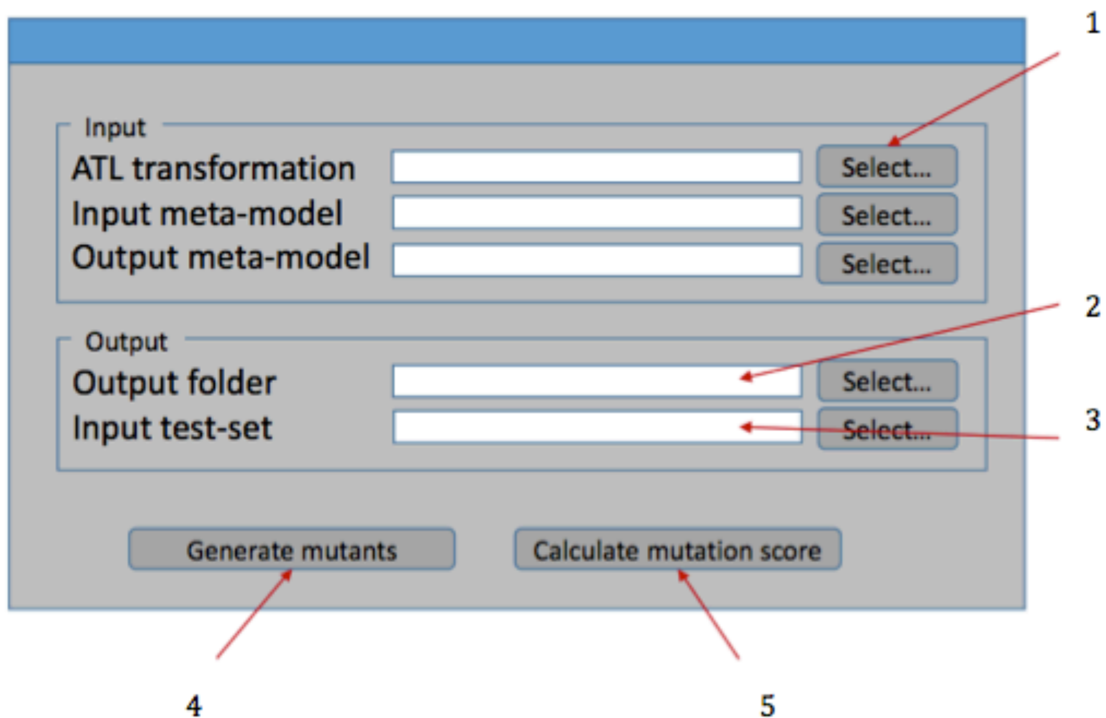

Figura 7. Maqueta de la pantalla principal

Tendríamos claramente dos zonas diferenciadas:

- En la parte de Input se especificarán los datos necesarios para la entrada de nuestra aplicación y que en nuestro caso son el fichero .atl con la transformación original, el fichero .ecore con el metamodelo de entrada y el fichero .ecore del metamodelo de salida.
  La localización de estos ficheros se podrá escribir directamente en el TextBox escribiendo la ruta del mismo o bien buscar el archivo mediante los botones de Select… (Punto 1 en la Figura 7).

- La parte de Output servirá para especificar, por un lado en el campo “Output folder” el directorio donde se quieren guardar todos los mutantes que se generen (punto 2), y por otro lado en el campo “Input test-set” el directorio que contiene todos los modelos de entrada del que queremos medir su calidad (punto 3).

El botón “Generate mutants” (punto 4) se habilitará sólo cuando se haya indicado un “Output folder”. Al pulsarlo, se generarán los mutantes en el directorio de salida especificado tal como se ha explicado en la sección 3.3. El contenido del campo “Input test-set” se ignorará en este caso.

El botón “Calculate mutation score” (punto 5) se habilitará sólo cuando se hayan rellenado los campos “Output folder” e “Input test-set”. Al pulsarlo, se generarán los mutantes y se calculará el porcentaje de mutación para el conjunto de modelos de prueba indicado siguiendo el algoritmo que se ha explicado en la sección 3.3. El resultado se guardará en un fichero .csv con la estructura que se muestra en la Figura 8.

	A	B	C	D	E
1		TOTAL	RSCC	ROCC	
2	number of mutants	12	7	5	
3	detected mutants	10	4	5	
4	non-detected m.	2	3	0	
5	MUTATION SCORE	83,33%	57,14%	100,00%	
6					
7					

Figura 8. Maqueta del fichero csv generado

En la Figura 8 se muestra cómo se visualizaría el contenido del fichero csv generado. En la segunda columna aparece el número total de mutantes generados (2ª fila), el número de mutantes detectados (3ª fila), el número de mutantes no detectados (4ª fila) y el porcentaje de mutación que indica la calidad del conjunto de casos de prueba (5ª fila). Las siguientes columnas muestras esos datos para cada tipo de mutación que haya sido generada.

5. Diseño

Una vez que se ha analizado qué es lo que se va a hacer vamos a ver cómo vamos a llevarlo a cabo, comenzando por describir e ilustrar mediante un diagrama, el flujo de información y qué operaciones se realizan en cada momento.

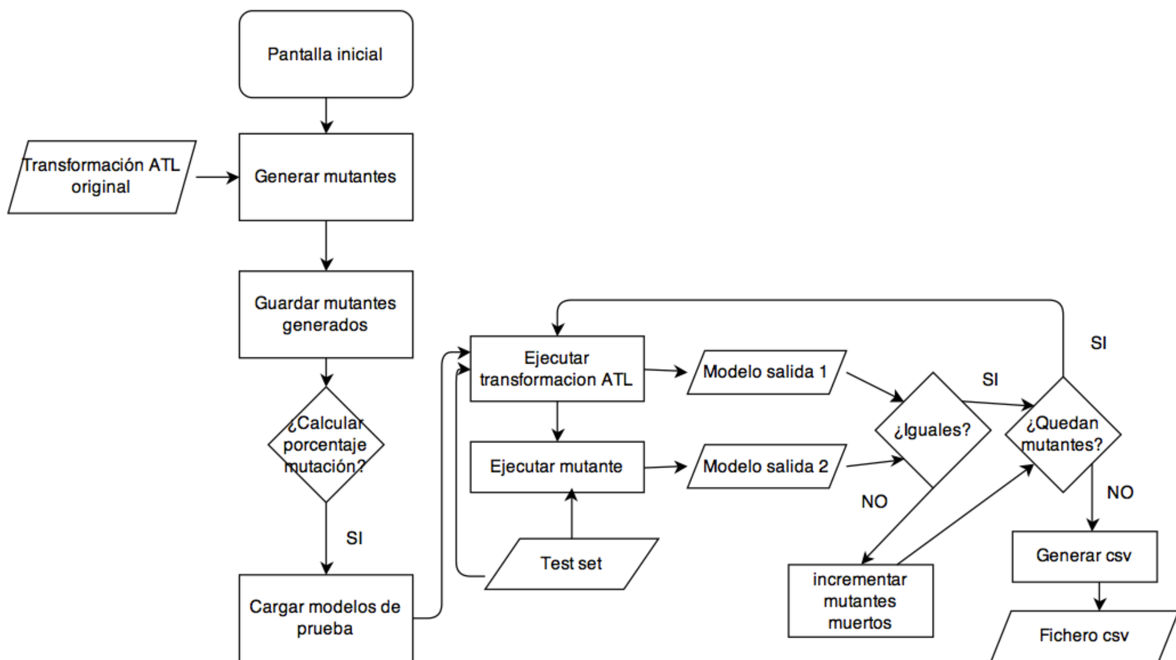

Figura 9. Diagrama de flujo de la aplicación

El diagrama de la Figura 10 muestra el diagrama de flujo de la aplicación. Inicialmente se partirá de la pantalla principal e, independientemente de la acción que se elija realizar, en ambos casos se realizará la generación de mutantes a partir de una transformación ATL original.

Una vez generados y guardadas las transformaciones mutantes, comprobamos la opción que se había elegido realizar. Si sólo se había elegido la opción de generar mutantes, el programa finaliza. Si por el contrario, se había elegido la opción de calcular los porcentajes de mutación, vamos al proceso que realiza este cálculo.

Primero se ejecuta la transformación ATL con un modelo de entrada (Test Set) y se obtiene un modelo de salida (Modelo salida 1), a continuación se ejecuta el mismo modelo de entrada con la transformación mutante, y se obtiene otro modelo de salida (Modelo salida 2); se comparan ambos modelos de salida y, si son iguales se vuelve a repetir el proceso con otro modelo de entrada, en caso de que sean distintos, se incrementa el numero e mutantes muertos y se vuelve a repetir el proceso habiendo descartado este último mutante. Cuando no queden más mutantes por estudiar, se calcula el porcentaje de mutación y se guardan los resultados en el fichero csv. De esta manera el programa finaliza su ejecución.

Una vez visto el flujo de información, vamos a ver cómo vamos a llevarlo a cabo en nuestra aplicación mediante el diagrama de clases correspondiente.

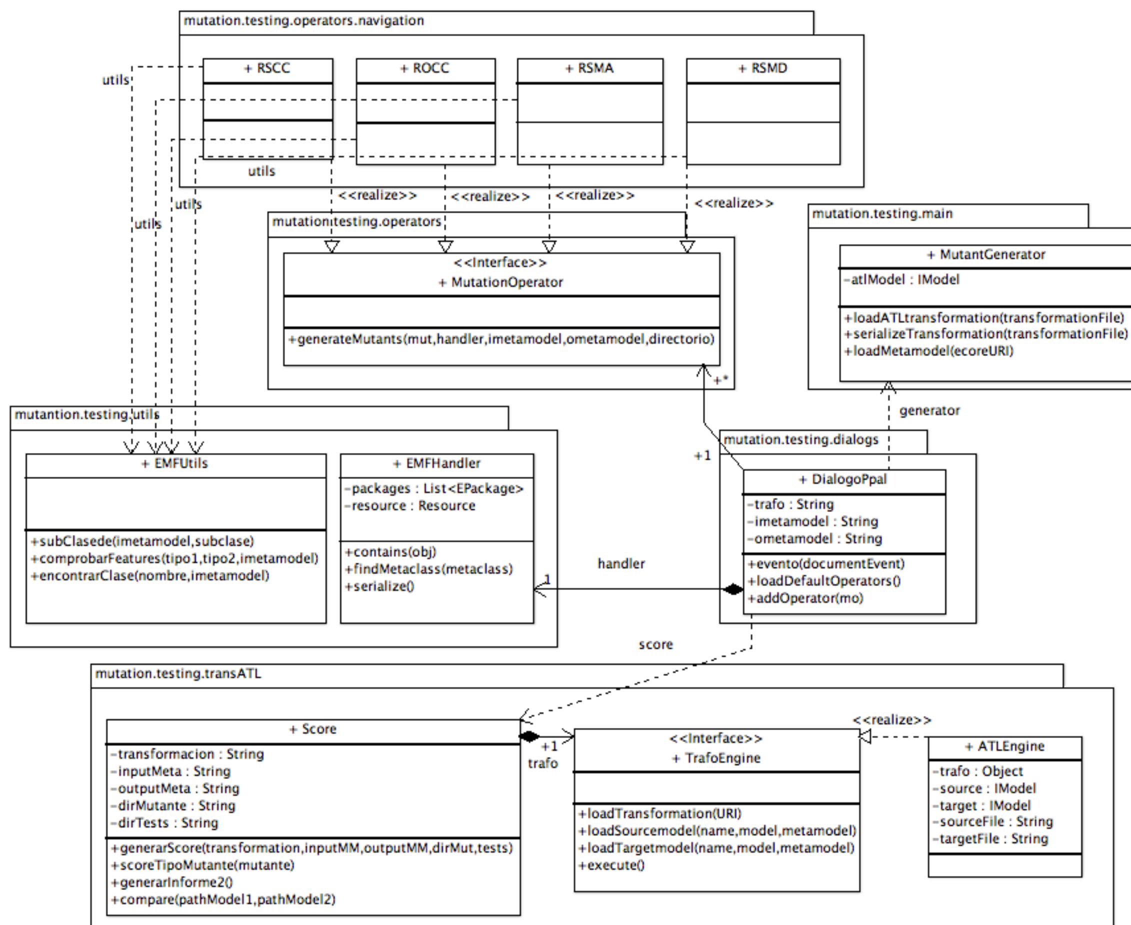

Figura 10. Diagrama de clases de la aplicación

La Figura 9 muestra el diagrama de clases de la aplicación completa que se ha desarrollado. Vamos a dar una breve descripción de lo que se realiza en cada una de estas clases.

- **MutationOperator**: es una interfaz que define el método de generación de mutantes “generateMutants”. Éste método recibe un objeto MutantGenerator con los métodos necesarios para cargar la transformación y los metamodelos en memoria, un objeto EMFHandler con métodos para manejar la transformación en memoria, el metamodelo de entrada, el metamodelo de salida y el directorio donde se guardarán los mutantes generados. El método genera los mutantes de la transformación y los guarda cada uno en un fichero, de esta manera el método no devuelve ningún valor. El objetivo de crear esta interfaz es que si alguien quiere usar otros operadores (además de RSCC, ROCC, RSMA y RSMD) solamente tenga que crear una clase que implemente esta interfaz y no haga falta que toque la clase principal “DialogoPpal”.
- **RSCC, ROCC, RSMA, RSMD**: cada una de estas clases se corresponde con un operador de mutación que implementa el método “generateMutants” de la interfaz MutationOperator. Se ha decidido implementar cada operador de mutación en una clase distinta para que si como trabajo futuro se quieren implementar mas tipos de mutación, bastaría con crear una nueva clase para este tipo de mutación, y el proyecto seguiría funcionando correctamente.
- **EMFUtils**: contiene métodos de utilidad para manejar los modelos EMF, proporcionando funcionalidades tales como comprobar si dos elementos de la transformación tienen las mismas propiedades. Estos métodos se utilizan en todas las clases del paquete “navigation”.
- **MutantGenerator**: contiene ciertos métodos de utilidad para llevar a cabo la generación de mutantes, tales como cargar en memoria una transformación, o un metamodelo.

- **DialogoPpal**: implementa la interfaz gráfica de usuario de la aplicación; es decir, muestra visualmente la aplicación y ejecuta los métodos necesarios para llevar a cabo la funcionalidad que se le ha especificado.
- **EMFHandler**: contiene métodos para manejar una transformación cargada en memoria, tales como obtener las propiedades de un elemento de la transformación o modificar un atributo.
- **TrafoEngine**: es una interfaz que define métodos implementados en la clase ATLEngine para el lenguaje de transformación ATL tales como la carga en memoria del modelo de entrada y la transformación o la ejecución de dicha transformación. Proporciona una interfaz común para la ejecución de transformaciones en distintos lenguajes. En concreto, ATLEngine la implementa para el lenguaje de transformación ATL.
- **ATLEngine**: implementa los métodos definidos en la interfaz TrafoEngine.
- **Score**: esta clase llevará a cabo la funcionalidad de calcular el porcentaje de mutación para un conjunto de modelos de prueba.

6. Implementación

Una vez descritos los detalles del análisis y el diseño de la aplicación, vamos a pasar a ver cómo se ha llevado a cabo esta implementación de pruebas de mutación.

6.1 Adaptación de las pruebas de mutación a transformaciones de modelos descritas en ATL

Ya se ha descrito en la sección 3.3 (Definición de las pruebas de mutación) cuál es el proceso que se sigue para llevar a cabo la generación de las pruebas de mutación; ahora vamos a ver cómo se ha realizado la implementación de este proceso. Las pruebas de mutación realizan dos funciones principales: por un lado la generación de las transformaciones mutantes y por otro lado el cálculo del

porcentaje de mutación. De esto, lo mas interesante es la generación de los mutantes.

6.1.1 Generación de una mutación ATL

Inicialmente se parte de una transformación escrita en lenguaje ATL en un fichero de texto con extensión .atl y de los metamodelos de entrada y salida escritos en un fichero con extensión .ecore. Mediante código generamos el modelo de la sintaxis abstracta de la transformación en un fichero .xmi con el aspecto que se muestra en la Figura 11. La sintaxis abstracta de ATL que se obtiene es de hecho un modelo conforme al metamodelo de ATL.

```
<source xsi:type="ocl:NavigationOrAttributeCallExp" location="45:4-45:10" name="type">
  <source xsi:type="ocl:VariableExp" location="45:4-45:5" referredVariable="/0/@elements.3/@inPattern/@ele
</source>
<arguments xsi:type="ocl:OclModelElement" location="45:23-45:35" name="DataType" model="/12"/>
</source>
<arguments xsi:type="ocl:OperatorCallExp" location="45:41-45:58" operationName="not">
  <source xsi:type="ocl:NavigationOrAttributeCallExp" location="45:45-45:58" name="multiValued">
    <source xsi:type="ocl:VariableExp" location="45:45-45:46" referredVariable="/0/@elements.3/@inPattern/@e
  </source>
</arguments>
</filter>
</inPattern>
</elements>
<elements xsi:type="atl:MatchedRule" location="57:1-75:2" name="MultiValuedDataTypeAttribute2Column">
  <outPattern location="62:2-74:4">
    <elements xsi:type="atl:SimpleOutPatternElement" location="63:3-66:4" varName="out">
      <type xsi:type="ocl:OclModelElement" location="63:9-63:18" name="Table" model="/16"/>
      <bindings location="64:4-64:39" propertyName="name">
        <value xsi:type="ocl:OperatorCallExp" location="64:12-64:39" operationName="+">
          <source xsi:type="ocl:OperatorCallExp" location="64:12-64:30" operationName="+">
            <source xsi:type="ocl:NavigationOrAttributeCallExp" location="64:12-64:24" name="name">
              <source xsi:type="ocl:NavigationOrAttributeCallExp" location="64:12-64:19" name="owner">
                <source xsi:type="ocl:VariableExp" location="64:12-64:13" referredVariable="/0/@elements.
                4/@inPattern/@elements.0"/>
              </source>
            </source>
            <arguments xsi:type="ocl:StringExp" location="64:27-64:30" stringSymbol="_"/>
```

Figura 11. Contenido del fichero xmi con la transformación

De esta manera podemos ver de forma mas concreta y visual todas los objetos que intervienen en la transformación así como las relaciones que existen entre ellos. Lo primero que hacemos es cargar la transformación original en memoria así como los metamodelos de entrada y salida, y recorremos la transformación para estudiar cada objeto que aparece es la misma. El proceso que

se va a describir es el mismo para todos los objetos de la transformación, así que se explicará solamente para uno de ellos.

Dado un objeto de la transformación original, lo que hacemos es localizar dicho objeto en el modelo de la sintaxis abstracta de la misma. Cada objeto de la transformación (*EObject*) guarda una referencia a su clase (*EClass*) y una localización (*location*) que indica la línea del fichero donde aparece dicho objeto. Dependiendo de la clase del objeto, éste puede definir distintas propiedades (*features*) que se corresponden con los atributos y relaciones de la clase. Por ejemplo, los objetos de tipo `NavigationOrAttributeCallExpr` (que se derivan de las expresiones de navegación de la transformación original) tienen como propiedades un nombre (*name*) y opcionalmente un objeto *source* que indicará el objeto que lo contiene. Para poder realizar un determinado cambio en la transformación original, tenemos que estudiar las propiedades del objeto y comprobar si el cambio se puede realizar o no. Si el cambio no se puede realizar, se pasara a estudiar el siguiente objeto de la transformación original, en caso de que se pueda realizar, generamos la mutación.

Una mutación en la transformación original implica un cambio de una de las propiedades del objeto; ya sea cambiar el nombre (name) o cambiar/eliminar el objeto con el que establece una relación (source), dependiendo del tipo de mutación que estemos haciendo se modificará una u otra cosa. Una vez se haya modificado el objeto en la transformación original cargada en memoria, guardamos esta transformación en otro fichero .atl, este fichero será una transformación mutante de la transformación original. El mutante generado cumple con la sintaxis de ATL y se puede compilar y ejecutar a pesar del error que se ha inyectado, ya que sólo se consideran errores que aseguran que la transformación sigue siendo sintácticamente válida tras el cambio. Esto es así, ya que este tipo de errores son más difíciles de detectar.

Cada transformación mutante tiene un único error; es decir por cada mutante sólo se realiza un cambio con respecto a la transformación original.

6.1.2 Generación del porcentaje de mutación

Cuando se han generado todos los ficheros .atl con las mutaciones posibles de la transformación original, se pasa a obtener el porcentaje de mutación de un conjunto de modelos de prueba. Para cada mutante, realizamos los mismos pasos: primero cargamos en memoria el mutante y la transformación original. Ejecutamos (mediante el método *execute* implementado en la clase TrafoEngine) la transformación mutante con un modelo de prueba descrito en un fichero de texto que puede tener extensión .model o extensión .xmi obteniendo un modelo de salida en el mismo formato, y la transformación original con el mismo modelo de prueba obteniendo otro modelo de salida.

Mediante el plugin de Eclipse EMFCompare, utilizamos una función para comparar dos ficheros de texto; en nuestro caso comparamos los dos ficheros que contienen los modelos de salida obtenidos de las dos ejecuciones. La función de comparar devuelve el número de diferencias que hay entre ambos ficheros, por lo que si esta función nos devuelve 0, significa que ambos ficheros son idénticos y en ese caso, volvemos a realizar las dos ejecuciones con otro modelo de entrada. Si la función de comparar devuelve un numero mayor que 0, significa que los ficheros no son iguales; en este caso desechamos el mutante que estamos ejecutando y pasamos al siguiente mutante generado para volver a realizar las ejecuciones.

De esta manera y, mediante la formula escrita en la sección 3.3, obtenemos el porcentaje total de mutación.

6.2 Tipos de mutaciones

Como se ha dicho anteriormente, una mutación no es más que una transformación a la que se le ha inyectado un error. Pero, este error se puede realizar de distintas formas, dependiendo del tipo de cambio que se introduzca en la transformación original. Los tipos de mutaciones que se proponen en [9] están divididos en tres grupos dependiendo de la implicación que suponga el cambio inyectado en el programa.

Las mutaciones de filtrado, que se trata de modificar una condición que deben cumplir los elementos de entrada para una regla determinada de la transformación.

Las mutaciones de creación, que consiste en la modificación de la creación de los elementos de los modelos de entrada y de salida de la transformación original.

Las mutaciones de navegación, que consiste en modificar la navegación entre dos objetos definida en la transformación original. Para este caso particular de mutaciones, siempre se debe garantizar que el cambio que se realiza sobre la transformación original es compatible con el tipado que define el metamodelo de entrada o de salida. Este tipo de mutaciones son las que se han implementado en la aplicación propuesta, las mutaciones de los otros dos grupos descritos se propone implementarlas como trabajo futuro. Vemos ahora mediante distintos ejemplos, los tipos de mutación que se pueden realizar con este tipo de cambio.

- **RSCC** (Relation to the Same Class Change): este tipo de mutación reemplaza una navegación de una clase A a una clase B por otra navegación distinta que también vaya de la clase A a la clase B.
  Sin embargo, se debe tener en cuenta la cardinalidad de la navegación a la clase B desde la clase A; es decir, para cambiar una relación por otra, en ambas relaciones la clase B debe tener la misma cardinalidad con respecta a

A. Esto se hace para asegurar que el mutante que se genera no contiene errores, ya que hay operadores que sólo son aplicables a colecciones (relaciones con cardinalidad 1 o mayor que 1) tales como *collet*, *select*, etc.

Esto queda mucho más claro de forma visual con un ejemplo.

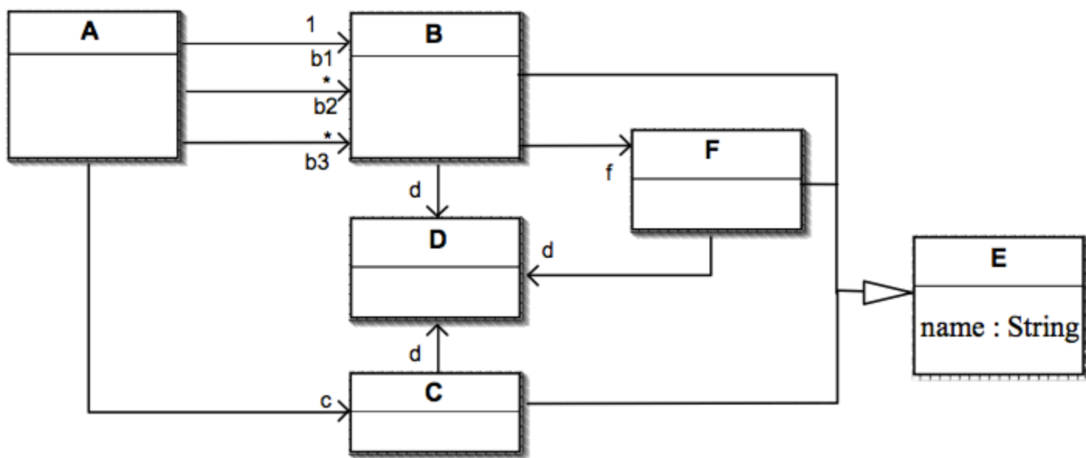

Figura 12. Ejemplo de metamodelo

La Figura 12 muestra un ejemplo de metamodelo. En este caso vemos que la clase A tiene tres relaciones distintas con la clase B. La relación b1 tiene cardinalidad 1, mientras que las relaciones b2 y b3 tienen cardinalidad múltiple, por lo que la navegación b1 no se podrá cambiar por ninguna otra navegación.

Sin embargo, la navegación b2 se podría cambiar por la navegación b3. Como ambas relaciones b2 y b3 tenían la misma cardinalidad, si un programador cometiera ese error en su transformación (poner b2 en vez de b3), sería difícil de detectar porque ambas relaciones llevan a la misma clase, de tal forma que la transformación "funcionaría" pero daría probablemente una solución incorrecta.

Vamos a ver ahora un ejemplo de este tipo de mutación en el ejemplo de metamodelo "Families" que pusimos al principio del documento (Figura 3). En este metamodelo tenemos cuatro navegaciones distintas: `familyFather`,

`familyMother`, `familySon` y `familyDaugther`, todas ellas tienen la misma clase de origen (`Member`) y la misma clase destino (`Family`); además, tienen la misma cardinalidad, todas con cardinalidad múltiple. Entonces en este caso, cualquier navegación se podría cambiar por cualquier otra; así se pueden realizar los siguientes cambios:

- `familyFather` → `familyMother/familySon/familyDaugther`
- `familyMother` → `familyFather/familySon/familyDaugther`
- `familySon` → `familyFather/familyMother/familyDaugther`
- `familyDaugther` → `familyFather/familyMother/familySon`

Cada posible cambio que se ha descrito se introduce siempre en la transformación. Los metamodelos de entrada y salida no son modificados en ningún caso.

Vamos a ver ahora cómo se ha insertado el cambio en la transformación ATL original.

```
helper context Families!Member def: isFemale(): Boolean =
  if not self.familyMother.oclIsUndefined() then
       true
  else
       if not self.familyDaughter.oclIsUndefined() then
            true
       else
            false
       endif
  endif;
```

Esta es una de las reglas (o helper en este caso) definidas en la transformación que realiza la conversión de un modelo Families en un modelo Persons. Como se ha explicado, uno de los posibles cambios sería cambiar la relación llamada familyMother por otra relación, en este caso por

familyFather, obteniendo la siguiente regla dentro de la nueva transformación mutante.

```
helper context Families!Member def: isFemale(): Boolean =
  if not self.familyFather.oclIsUndefined() then
       true
  else
       if not self.familyDaughter.oclIsUndefined() then
            true
       else
            false
       endif
  endif;
```

Este cambio inyectado en la transformación original es lo que llamamos un mutante; un cambio que genera una transformación mutante completamente distinta.

Vemos ahora cómo se realiza esta mutación en la implementación de nuestra aplicación. En primer lugar obtenemos el tipo, la cardinalidad y la clase a la que pertenece el objeto `familyMother`, que es el objeto que queremos modificar, del metamodelo de entrada. Una vez tenemos estos datos del objeto que se quiere cambiar, recorremos la transformación ATL original y buscamos todos los objetos que pertenezcan a la misma clase y que tengan el mismo tipo. Si el objeto `familyMother` tiene cardinalidad -1 o cardinalidad mayor que 1, se buscarán los objetos que además de pertenecer a la misma clase y de tener el mismo tipo que el objeto `familyMother`, además tengan cardinalidad -1 o cardinalidad 1, y dichos objetos podrán cambiarse por el objeto `familyMother` para obtener una mutación (cada mutación implica un único cambio). Si por el contrario, el objeto `familyMother` tiene cardinalidad 1, se buscarán los objetos que además de pertenecer a la misma clase y de tener el mismo tipo que el objeto `familyMother`, además tengan cardinalidad 1, y podrán ser cambiados por el objeto original `familyMother`.

De manera general, lo que se hace es buscar todos los objetos de tipo `NavigationOrAttributeCallExpr` que aparecen en la transformación de la Figura 11, y cambiarlo por otro objeto también de tipo `NavigationOrAttributeCallExpr`, verificando que se cumplan las condiciones para poder aplicar el cambio (descritas al comienzo de la sección).

- **ROCC** (Relation to Another Class Change): este tipo de mutación es muy similar a la mutación RSCC, salvo que en este caso la navegación original se reemplaza por otra cuya clase de destino sea distinta a la clase de destino de la navegación original. Es decir, dada una relación de la clase A a la clase B, dicha relación se podrá sustituir por una relación de la clase A a otra clase C. Para asegurarnos de que el mutante que se genera se ejecutará correctamente, la nueva clase a la que se navega debe pertenecer a la misma clase o a una superclase de la clase a la que se navegaba originalmente; además, el tipo de ambas clases debe ser distinto, al contrario que pasaba en las mutaciones de tipo RSCC en el que debían de tener el mismo tipo. También para este tipo de mutación, la nueva clase tiene que tener las mismas propiedades (*features*) o mas, que la clase a la que se navega originalmente.

En el ejemplo de metamodelo expuesto en la Figura 12, se podría cambiar la relación b1 por la relación c, debido a que ambas relaciones tienen distinta clase de destino; la primera navega de A a B y la segunda navega de A a C y a que B tienen las mismas propiedades que C; sin embargo no podríamos realizar el cambio de c por b1, debido a que B define la relación f, y C no define esta relación. Esta condición de que deben tener las mismas propiedades hace que la mutación sea mas restrictiva de manera que sea mas difícil para un modelo de prueba de detectar este error.

En el ejemplo del metamodelo Families, no podríamos realizar ninguna mutación de este tipo, debido a que todas las navegaciones

existentes tienen la misma clase de origen (`Member`) y la misma clase de destino (`Family`).

Como este metamodelo no nos sirve para ver una posible mutación de este tipo, mostramos otro ejemplo de metamodelo de entrada:

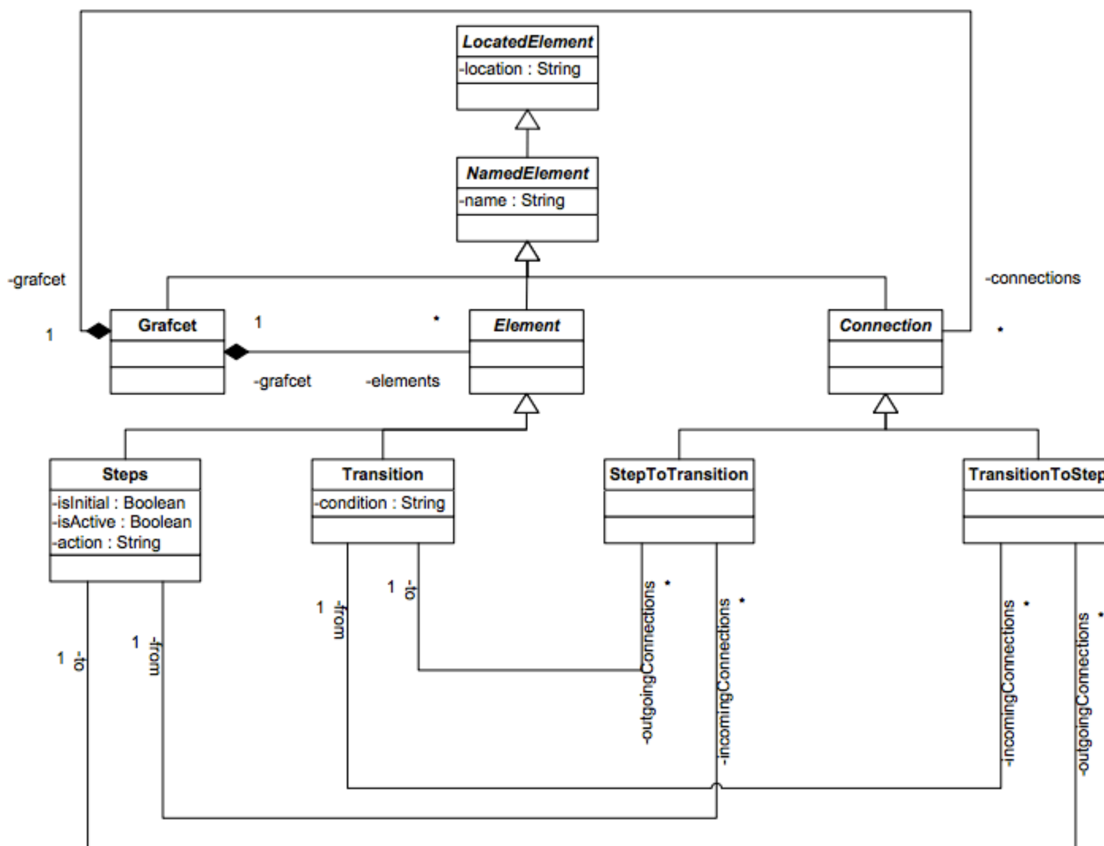

Figura 13. Metamodelo Grafcet

En el ejemplo de metamodelo de la Figura 13, encontramos una posible mutación de tipo ROCC; la relación connections se puede reemplazar por la navegación elements, debido a que ambas relaciones tienen una misma clase de origen `Grafcet` y distintas clase de destino; la primera tiene como clase destino Connection y la segunda `Element`.

Igual que hemos hecho en el caso de RSCC, vamos a poner un ejemplo de este error insertado en la transformación original que

transforma un modelo de entrada de tipo `Grafcet` en otro modelo de salida de tipo `PetriNets`.

```
rule PetriNet {
	from
		g : Grafcet!Grafcet
	to
		p : PetriNet!PetriNet
		(
			location <- g.location,
			name <- g.name,
			elements <- g.elements,
			arcs <- g.connections
		)
}
```

Esta es una regla de la transformación original; veamos ahora cómo sería la regla con el error inyectado.

```
rule PetriNet {
	from
		g : Grafcet!Grafcet
	to
		p : PetriNet!PetriNet
		(
			location <- g.location,
			name <- g.name,
			elements <- g.connections,
			arcs <- g.connections
		)
}
```

Como se ha explicado antes, el error consistía en modificar la relación elements por la relación `connections`; de esta forma esta nueva regla forma parte de la transformación mutante.

- **RSMA** (Relation Secuence Modification with Addition): este tipo de mutación añade una navegación extra a continuación de la navegación original, siempre que ambas relaciones tengan el mismo tipo y las mismas propiedades (*features*); además, tanto la clase a la que se le añade la navegación nueva, como la clase a la que se navega en la navegación añadida deben tener cardinalidad 1. Esto es, en el ejemplo de metamodelo de la Figura 12, a la navegación de c que va de A → C, se le añade la navegación d de manera que ahora la navegación va de A → D. Es decir, que todas las referencias de tipo A → C se van a cambiar en la transformación original por referencias de tipo A → C → D.

Con los metamodelos de `Families` y `Grafcet` no tenemos un ejemplo de posible mutación de este tipo, por lo que vamos a poner otro ejemplo de metamodelo en el que sí se pueda realizar.

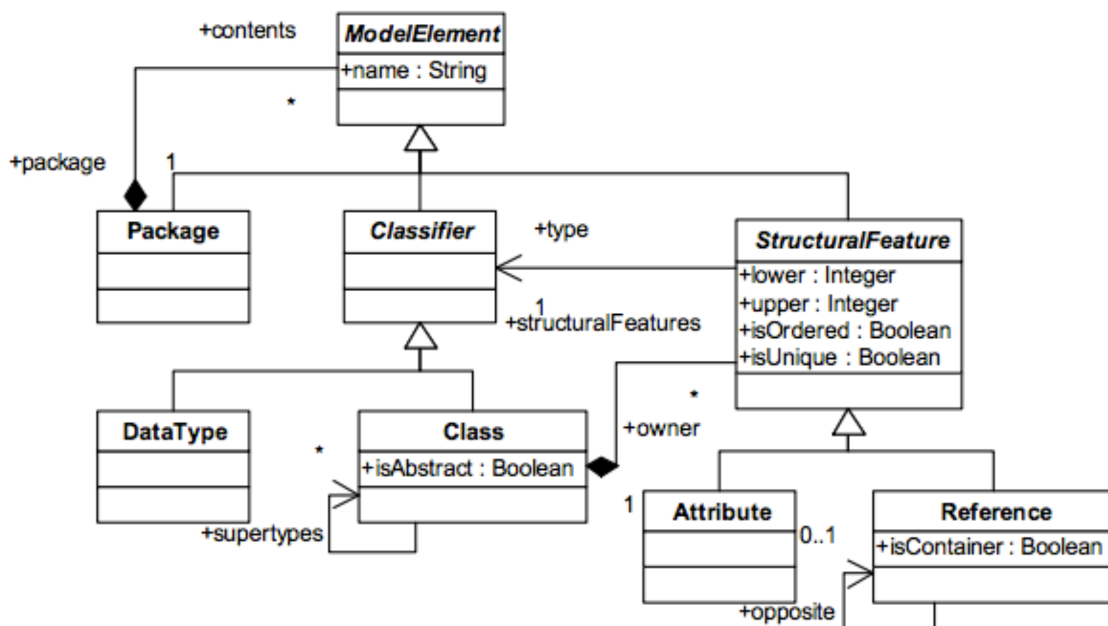

Figura 14. Metamodelo KM3

En el ejemplo de la Figura 14, podríamos añadir una navegación extra a la referencia `opposite`. Esta referencia navega desde la clase

Reference a la propia clase Reference, por este motivo se puede añadir una referencia mas y reemplazar la navegación Reference → (opposite) → Reference por la navegación Reference → (opposite) → (opposite) → Reference. Pongo entre paréntesis la navegación para que no se confunda con la clase.

Vemos ahora cómo se lleva este cambio a la transformación ATL que se ha estudiado para este trabajo. Mostramos primero la regla en la que vamos a introducir este error dentro de la transformación original que realiza el cambio de un modelo de entrada de tipo KM3 a un modelo de salida de tipo DOT.

```
helper def: relationsList: Sequence(
   TupleType(ref: KM3!Reference, opposite : KM3!Reference)) =
   let references: Sequence(KM3!Reference) =
   KM3!Reference.allInstances()->
   reject( e | e.opposite.oclIsUndefined()) in references->iterate( e;
       acc: Sequence(TupleType(ref: KM3!Reference, opposite:
       KM3!Reference)) =
        Sequence{} |
               if acc->excludes(Tuple{ref = e, opposite = e.opposite})
   then
                     if acc->excludes(Tuple{ref = e.opposite, opposite =
                     e}) then
                            if e.opposite.isContainer then
                                  acc->append(Tuple{ref = e, opposite =
                                  e.opposite})
                            else
                                  acc->append(Tuple{ref = e.opposite,
                                  opposite = e})
                            endif
                      else
                            acc
                      endif
               else
                      acc
               endif);
```

Esto describe una regla dentro de la transformación inicial. Vemos ahora dónde exactamente de insertaría el error que hemos descrito mas arriba.

```
helper def: relationsList: Sequence(
   TupleType(ref: KM3!Reference, opposite : KM3!Reference)) =
   let references: Sequence(KM3!Reference) = KM3!Reference.allInstances()
   ->
   reject( e | e.opposite.opposite.oclIsUndefined()) in references-
   >iterate( e;
       acc: Sequence(TupleType(ref: KM3!Reference, opposite:
       KM3!Reference)) =
        Sequence{} |
               if acc->excludes(Tuple{ref = e, opposite = e.opposite})
   then
                     if acc->excludes(Tuple{ref = e.opposite, opposite =
                     e}) then
                              if e.opposite.isContainer then
                                    acc->append(Tuple{ref = e, opposite =
                                    e.opposite})
                              else
                                    acc->append(Tuple{ref = e.opposite,
                                    opposite = e})
                              endif
                      else
                              acc
                      endif
               else
                      acc
               endif);
```

El mutante que se había descrito consistía en añadir una relación extra al final de la navegación; en el ejemplo expuesto se trataba de añadir la navegación `opposite` detrás de la navegación original. En este caso vemos cómo se ha añadido esta navegación y el sitio donde se ha insertado el error. Pero en esta regla hay mas sitios donde aparece la navegación `e.opposite`, y por lo tanto saldrían mas mutantes, recordamos que cada mutante contiene un único cambio. En concreto, saldrían dos mutantes ya que hay dos posibles mutaciones de `e.opposite`.

En este proyecto en particular, se ha realizado una pequeña modificación a este tipo de mutación; en concreto no sólo se ha realizado la inserción de una relación extra detrás de la navegación actual, si no que

también se ha considerado insertar la relación adicional delante de la navegación que teníamos al principio, siempre y cuando los tipos y características de ambas relaciones sean idénticos; de esta manera obtenemos más transformaciones mutantes de este tipo.

- **RSMD** (Relation Secuence Modification with Deletion): este tipo de mutación es muy similar a RSMA, salvo que en este caso en lugar de añadir una navegación extra, lo que se hace es eliminar la navegación final. En este caso, las condiciones para poder realizar la mutación son las mismas que para las mutaciones de tipo RSMA; es decir, tanto la clase origen de la navegación original como la clase destino (que se quiere eliminar) de la navegación original deben tener el mismo tipo y las mismas propiedades (*features*) y, además, ambas clases deben tener cardinalidad 1. Por ejemplo en el caso del metamodelo de la figura 14, podríamos eliminar la ultima navegacion de `Reference` → (`opposite`). Es decir eliminamos la referencia `opposite`. Esto se puede realizar debido a que la clase a la que se navega (`Reference`) es la misma que la clase desde la que se navega, que también es `Reference`. Además, como el objeto `Reference` define la propiedad `name`, la navegación resultante tiene sentido y el cambio se puede realizar.

  Vamos a ver ahora cómo realizamos esta modificación en una regla de la transformación de un modelo de tipo KM3.

```
rule Reference2Arc {
from
	r: KM3!Reference ( ...)
to
	out: DOT!DirectedArc (
		fromNode <- r.owner,
		...),
	ArcHeadLabel : DOT!SimpleLabel (
		...),
	ArcTailLabel : DOT!SimpleLabel (
		content <- r.opposite.name +
```

```
                r.opposite.getMultiplicity() +
                        if r.opposite.isOrdered then
                                '{ordered}'
                        else
                                ''
                        endif
                ),
```

Esta regla, contiene la navegación que acabamos de explicar. Introducimos ahora el error para generar la transformación mutante.

```
rule Reference2Arc {
from
        r: KM3!Reference ( ...)
to
        out: DOT!DirectedArc (
                fromNode <- r.owner,
                ...),
        ArcHeadLabel : DOT!SimpleLabel (
                ...),
        ArcTailLabel : DOT!SimpleLabel (
                content <- r.name + r.opposite.getMultiplicity() +
                        if r.opposite.isOrdered then
                                '{ordered}'
                        else
                                ''
                        endif
                ),
```

Podemos ver cómo se ha eliminado la navegación `opposite` y cómo de esta manera hemos realizado otra transformación mutante mas.

Con esto quedan descritas todas y cada una de las posibles mutaciones de navegación que podemos realizar sobre una transformación original. Aunque las mutaciones de filtrado y de creación no han sido implementadas en este caso, gracias al diseño extensible que se ha realizado, sería fácil incorporarlas al proyecto sin tocar lo que ya hay desarrollado.

7 Caso de estudio

Como bien hemos transmitido en este documento, las pruebas son parte fundamental de cualquier aplicación. Vamos ahora a ver cómo funciona la aplicación propuesta y cuáles son las funcionalidades y objetivos de la misma. Nuestras pruebas se realizarán sobre una nueva transformación sacada del repositorio público de transformaciones [1]. En concreto, trataremos la transformación que a partir de un modelo de entrada de tipo `Class`, genera un modelo de salida de tipo `Relational`. El metamodelo de entrada Class (Figura 15) y el metamodelo de salida `Relational` (Figura 16) se muestran a continuación:

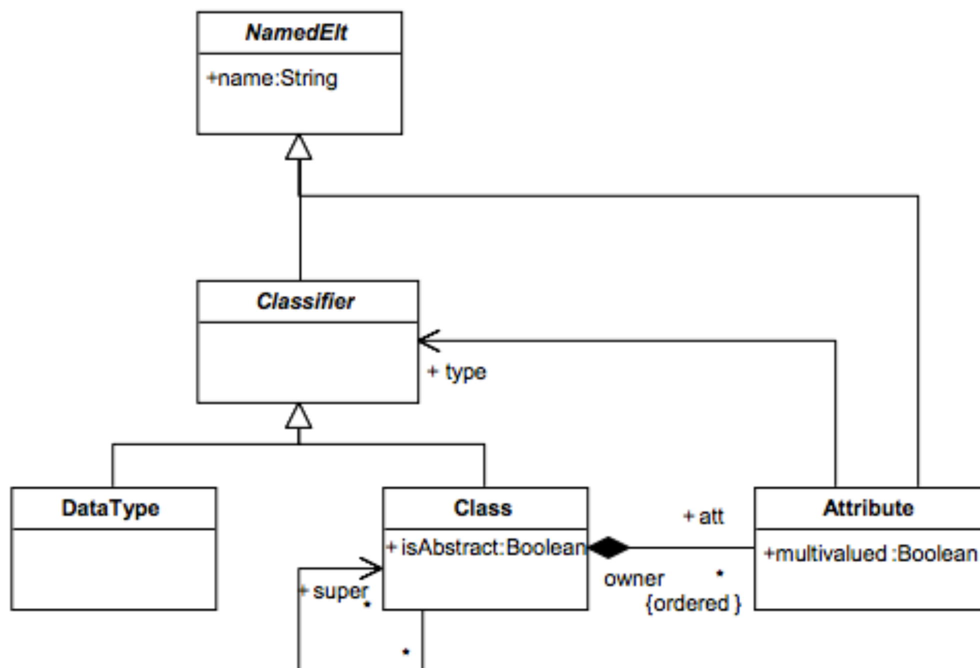

Figura 15. Metamodelo Class

[1] http://www.eclipse.org/atl/atlTransformations/

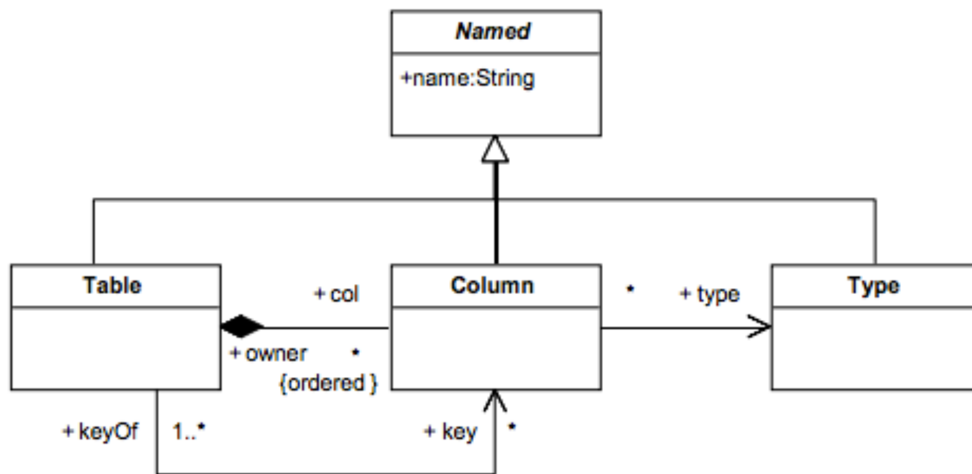

Figura 16. Metamodelo Relational

Este va a ser nuestro punto de partida. Un metamodelo de origen llamado Class.ecore y un metamodelo de salida llamado Relational.ecore.

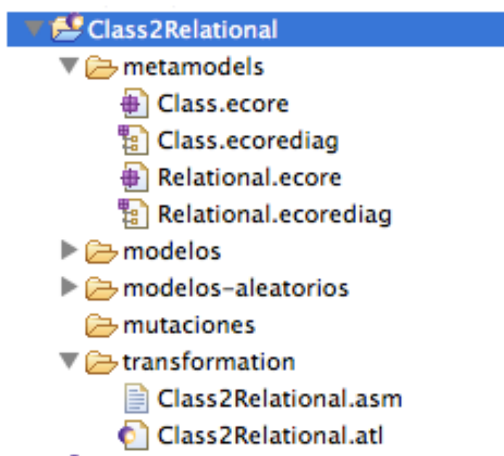

Figura 17. Estructura del proyecto Class2Relational

Dentro de la carpeta transformación está el fichero Class2Relational.atl que es donde está implementada la transformación de modelos. Las carpetas de "modelos" y "modelos-aleatorios" contienen todos los modelos de prueba que se van a utilizar para las pruebas. Los 45 modelos que se encuentran en la carpeta "modelos" se han generado usando el enfoque que se presenta en el articulo [16], mientras que los 45 modelos de la carpeta "modelos-aleatorios" han sido

generados, como su propio nombre indica, de forma aleatoria. Nuestro objetivo será comparar los porcentajes de mutación que genera cada uno de ellos. Y, decidir así que conjunto de casos de prueba es mas completo y eficaz.

7.1 Generación de mutantes

Vamos a generar primero todos los mutantes posibles para esta transformación de modelos. Nada más ejecutar la aplicación, lo primero que se muestra es la pantalla principal para seleccionar todos los parámetros que se necesitan para llevar a cabo la ejecución correcta del programa.

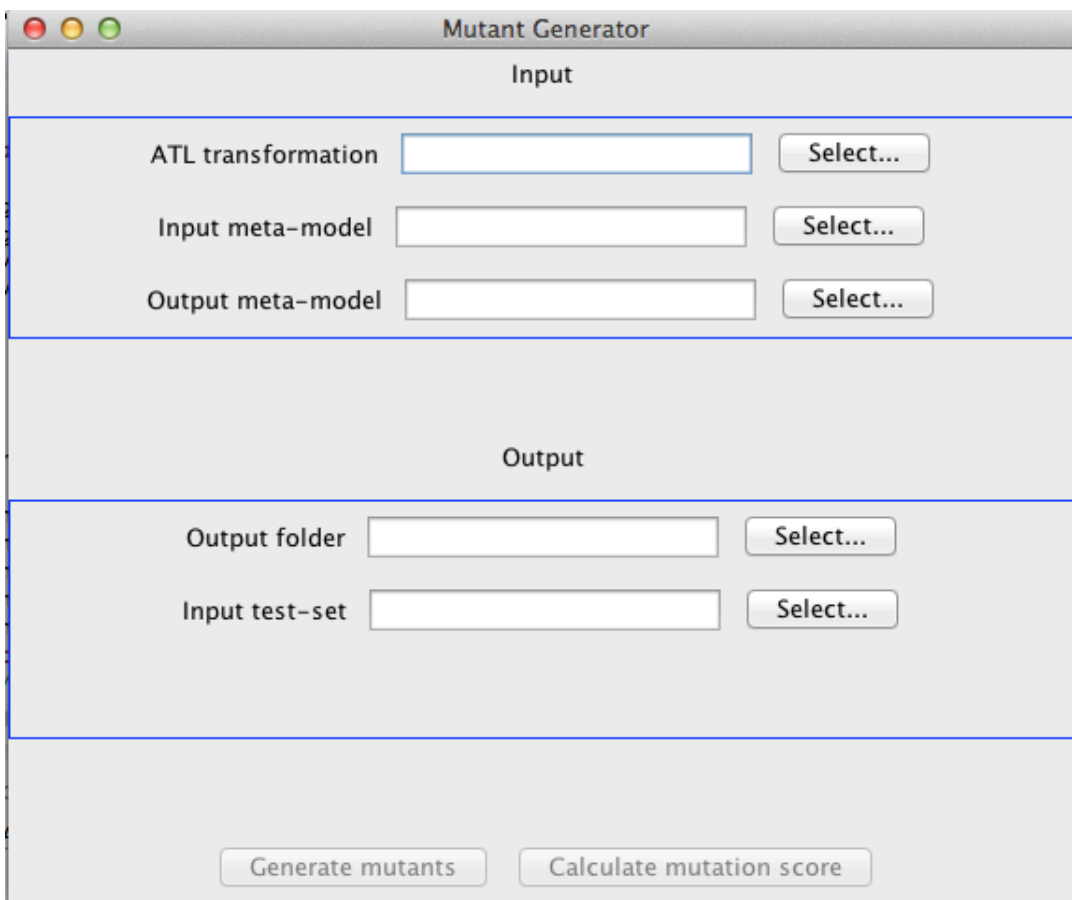

Figura 18. Pantalla de inicio de la aplicación

Inicialmente los botones de "Generate mutants" y "Calculate mutation score" aparecen inhabilitados y, sólo se habilitarán cuando hayamos introducido

los datos necesarios para realizar estas operaciones, como se ha especificado en el análisis de la aplicación (sección 4).

Vamos a seleccionar los elementos del apartado Input, como son la transformación ATL, el metamodelo de entrada y el metamodelo de salida. Esto los podemos hacer de dos formas, como ya se ha dicho, poniendo la ruta del fichero a mano en el TextBox, o bien pulsamos sobre el botón “Select…”.

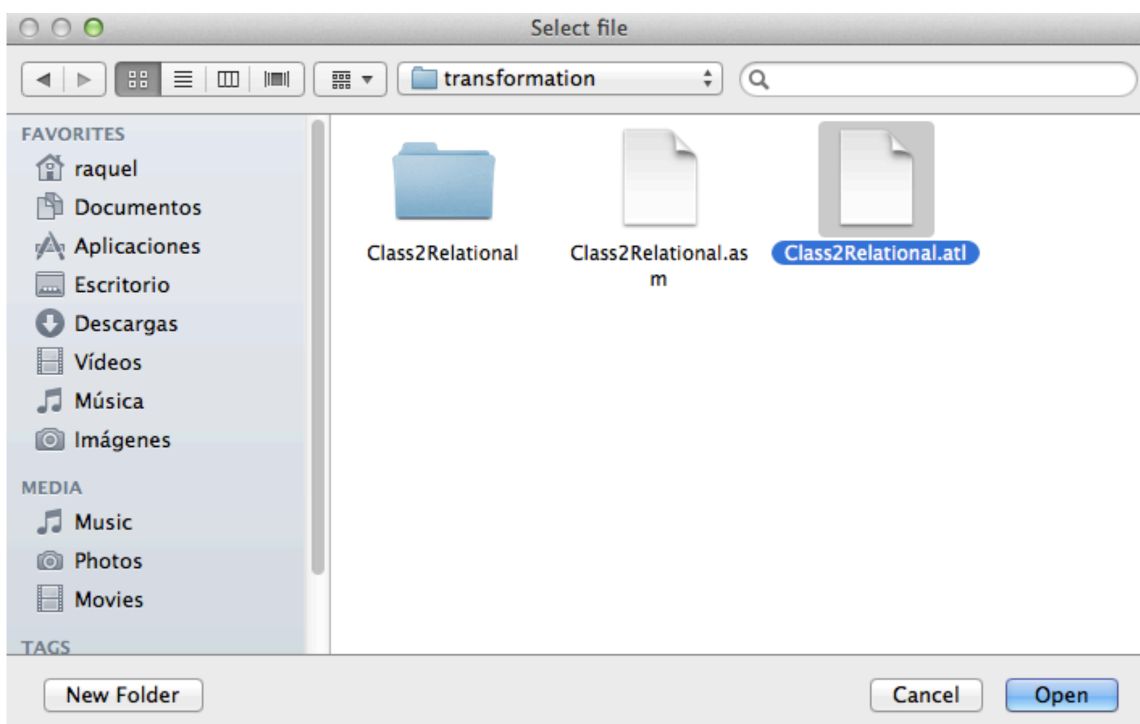

Figura 19. Pantalla de selección de archivos de entrada

Una vez localizado el archivo de la transformación, lo abrimos y nos habrá escrito la ruta completa del fichero en el TextBox correspondiente a la transformación ATL. Realizamos los mismos pasos para introducir el metamodelo de entrada y el de salida.

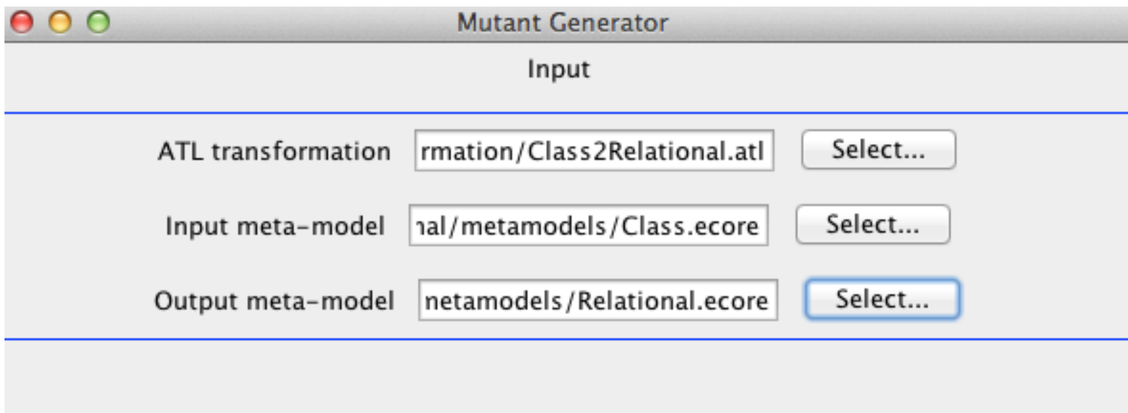

Figura 20. Insertar ruta de los archivos

Vamos a introducir ahora los datos del apartado Output,; de nuevo podemos introducir la ruta directamente en el TextBox o hacer uso del botón "Select..." para seleccionar el directorio donde se quieren guardar los mutantes y el directorio donde se encuentran el conjuntos de modelos de prueba.

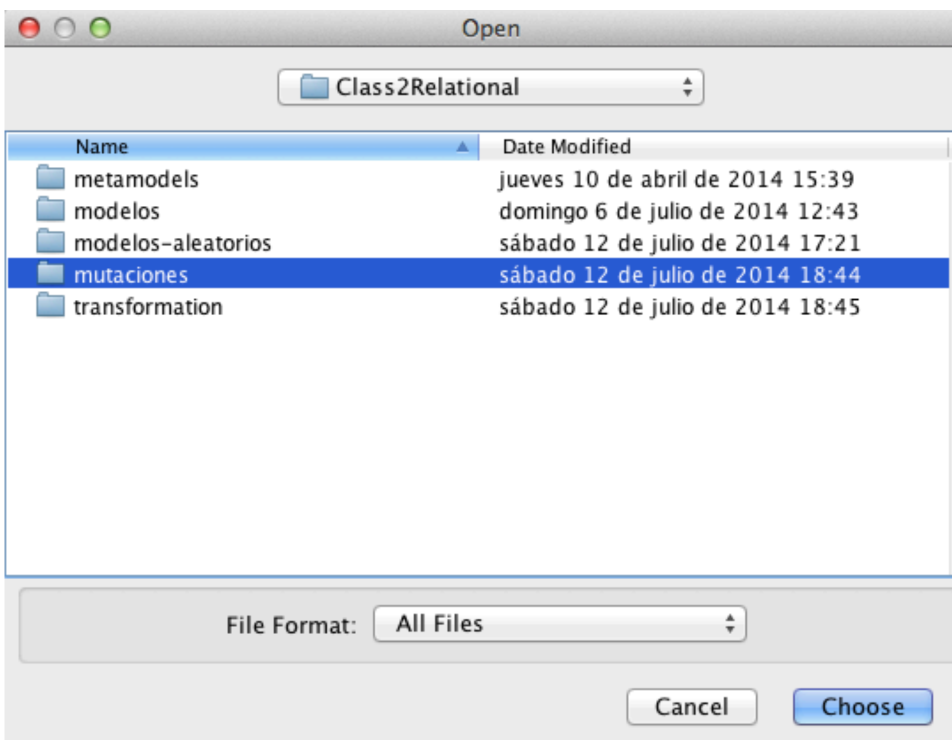

Figura 21. Pantalla de selección de directorio

Una vez completados todos estos campos, vemos cómo el botón para generar mutantes se activa.

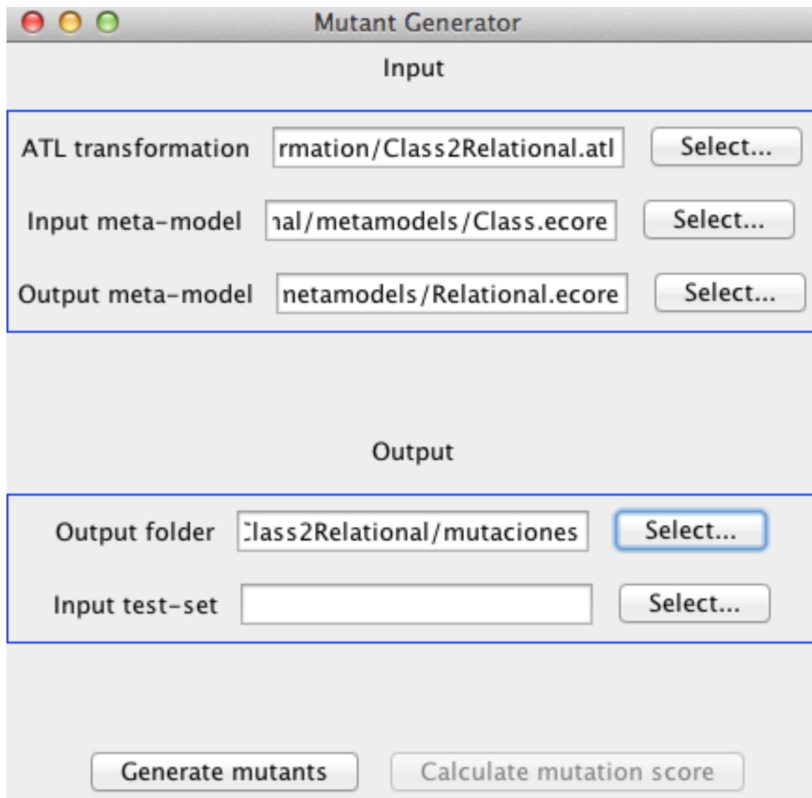

Figura 22. Pantalla de generación de mutantes

Entonces, ya podemos generar los mutantes. Una vez pulsemos "Generate Mutants", si los mutantes se han generado correctamente, nos aparecerá el siguiente mensaje:

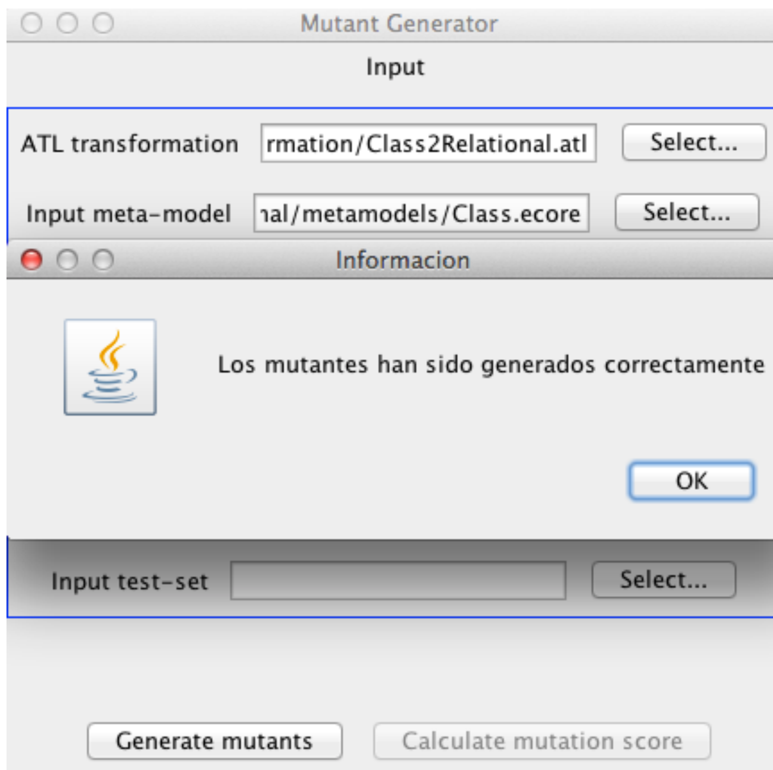

Figura 23. Pantalla de verificación

Y una vez pulsado el “OK”, el programa se cerrará. En la carpeta “Class2Relational” se habrán generado los mutantes que muestra la Figura 24:

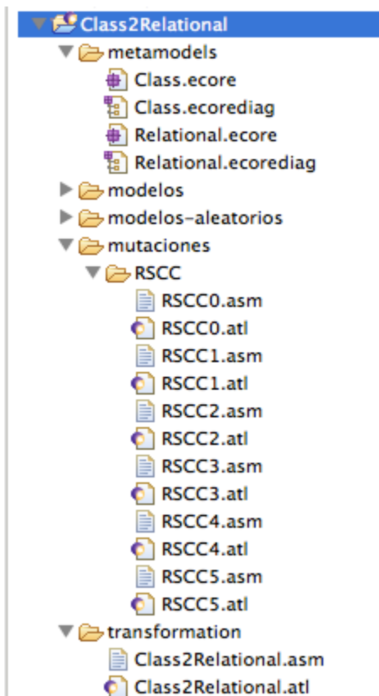

Figura 24. Estado del proyecto Class2Relational

Como podemos ver, para este caso se han generado sólo mutantes de tipo RSCC, esto se debe a que en esta transformación no existen posibles mutaciones del resto de tipos que han sido descritos. En concreto, se han generado 6 mutaciones de la transformación original. Vamos a ver una de ellas, por ejemplo la mutación RSCC2.atl y vamos a compararla con la transformación original Class2Relational.atl.

```
rule MultiValuedDataTypeAttribute2Column {
    from
        a : UML!Attribute (
            a.type.oclIsKindOf(UML!DataType) and
            a.multiValued
        )
    to
        out : Rel!Table (
            name <- a.owner.name + '_' + a.name,
            col <- Sequence {id, value}
        ),
        id : Rel!Column (
            name <- a.owner.name + 'Id',
            type <- thisModule.objectIdType
```

```
		),
		value : Rel!Column (
			name <- a.name,
			type <- a.type
		)
}
```

Esta es una de las reglas de la transformación original. Vemos ahora esta misma regla en la transformación mutante RSCC2.atl.

```
rule MultiValuedDataTypeAttribute2Column {
	from
		a : UML!Attribute (
			a.type.oclIsKindOf(UML!DataType) and
			a.multiValued
		)
	to
		out : Rel!Table (
			name <- a.owner.name + '_' + a.name,
			col <- Sequence {id, value}
		),
		id : Rel!Column (
			name <- a.owner.name + 'Id',
			type <- thisModule.objectIdType
		),
		value : Rel!Column (
			name <- a.name,
			type <- a.owner
		)
}
```

Vemos cómo el error que ha sido inyectado en este mutante ha sido modificar la navegación `a.type` original y cambiarla por `a.owner`. Esta modificación se puede realizar porque 'a' es un `Attribute`, y la nueva relación `owner` pertenece al tipo `Class`, que es una subclase de `Classifier` que contiene la relación original `type` (ver Figura 15). Recordamos que para realizar una cierta mutación se deben cumplir este tipo de requisitos entre las clases afectadas en una relación.

7.2 Calcular porcentajes de mutación

Vamos a ver ahora la otra funcionalidad de la aplicación, que es calcular los porcentajes de una mutación de acuerdo con lo explicado en la sección 3.3 (Definición de las pruebas de mutación). En concreto, por cada uno de los 6 mutantes que hemos generado, se ejecutarán todos los modelos de prueba con la transformación original y luego se ejecutarán con uno de estos mutantes, y se compararán los modelos de salida obtenidos. Si los modelos son iguales, se continúa con el siguiente modelo de prueba; si son distintos, el mutante utilizado se desecha, ya que el modelo de prueba ha sido capaz de detectar la mutación.

Entonces, volvemos a ejecutar la aplicación, pero esta vez rellenamos también el campo de Input test-set, donde pondremos el directorio donde se encuentran los modelos de prueba que queremos estudiar. En nuestro caso, vamos a poner primero el directorio de la carpeta “modelos”, que son los modelos generados de acuerdo con el documento [16]. Después de esto, quedan rellenos todos los campos como se muestra en la Figura 25.

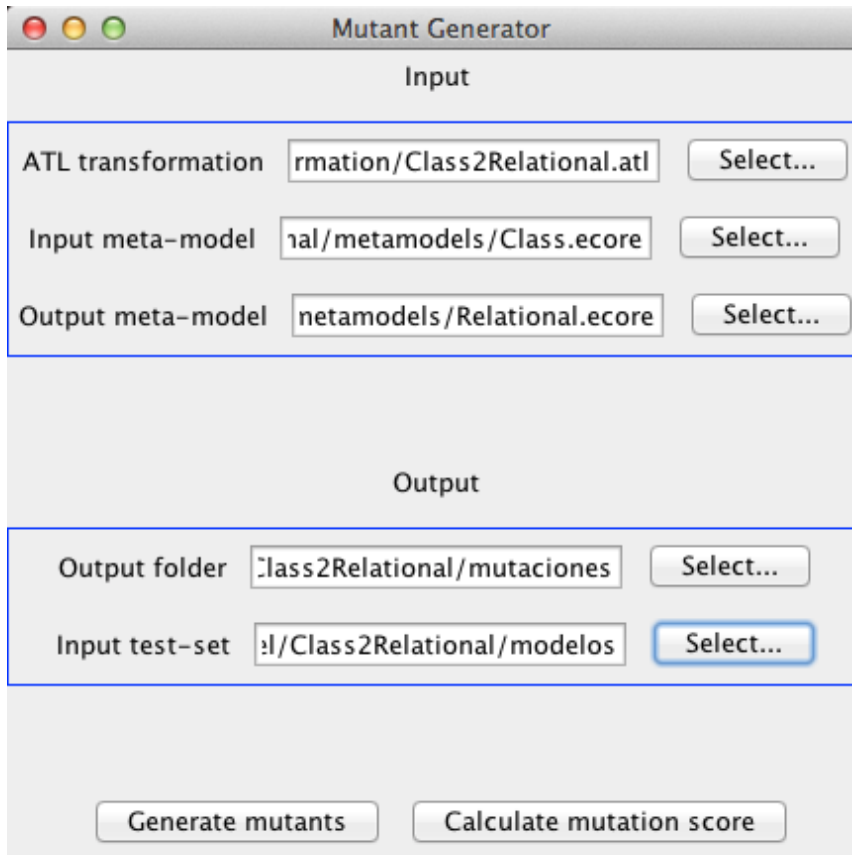

Figura 25. Pantalla de cálculo de porcentajes

Vemos que el botón "Calculate mutation score" se ha activado. Lo pulsamos entonces para generar el porcentaje de mutación. Si todo ha ido correctamente, el programa finalizará automáticamente después de esto. En la carpeta de nuestro proyecto Class2Relational, veremos elementos nuevos.

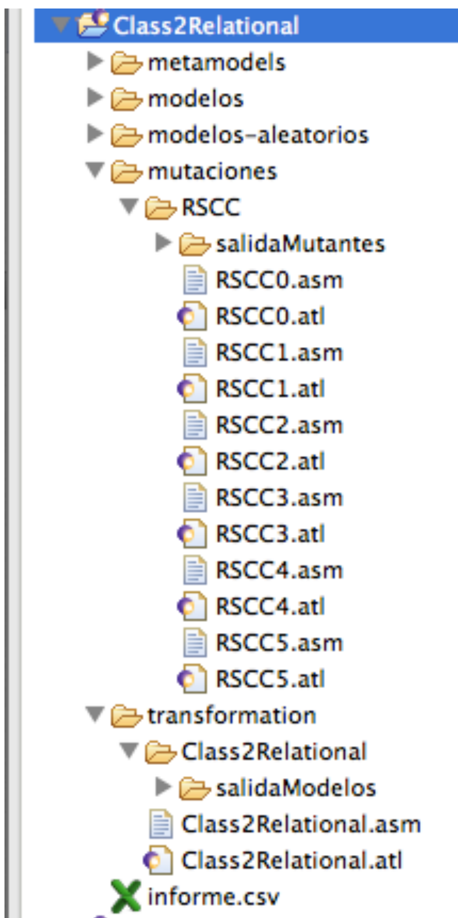

Figura 26. Ficheros del proyecto Class2Relational

En la Figura 26, vemos que se han creado las carpetas "salidaMutantes" y "salidaModelos", en ellas están todos los modelos de salida resultado de ejecutar, en el primer caso cada mutante con cada modelo de prueba, y en el segundo la transformación original con cada modelo de prueba.

También, vemos un documento "informe.csv" donde se han guardado los resultados obtenidos y el porcentaje de mutación para este conjunto de casos de prueba.

	TOTAL	RSCC	ROCC	RSMA	RSMD
number of mutants	6.0	6	0	0	0
detected mutants	4.0	4	0	0	0
non-detected mutants	2	2	0	0	0
MUTATION SCORE	0.6666667	0.6666667	0.0	0.0	0.0

Figura 27. Vista del fichero csv de salida para modelos de entrada

En la Figura 27 podemos ver estos resultados. Dado que para esta transformación sólo hay mutaciones de tipo RSCC, el resto de mutaciones permanecen con porcentaje 0.

En este caso, el número de mutantes generados es 6 (como hemos visto en el apartado anterior), el número de mutantes detectados es 4, esto quiere decir que 4 de los 6 mutantes creados han sido detectados por el conjunto de casos de prueba que hemos utilizado; sin embargo hay 2 mutantes no se han detectado. El porcentaje total de mutación se calcula mediante la fórmula explicada en la sección 3.3:

$$\text{Porcentaje de mutación} = \frac{\text{número de mutantes "muertos"}}{\text{número total de mutantes generados}}$$

Vamos ahora a ver los porcentajes de mutación obtenidos si volvemos a ejecutar la aplicación pero esta vez cambiamos el conjunto de casos de prueba y ponemos los modelos que se generaron automáticamente, es decir, los que están en la carpeta “modelos-aleatorios” (ver Figura 28).

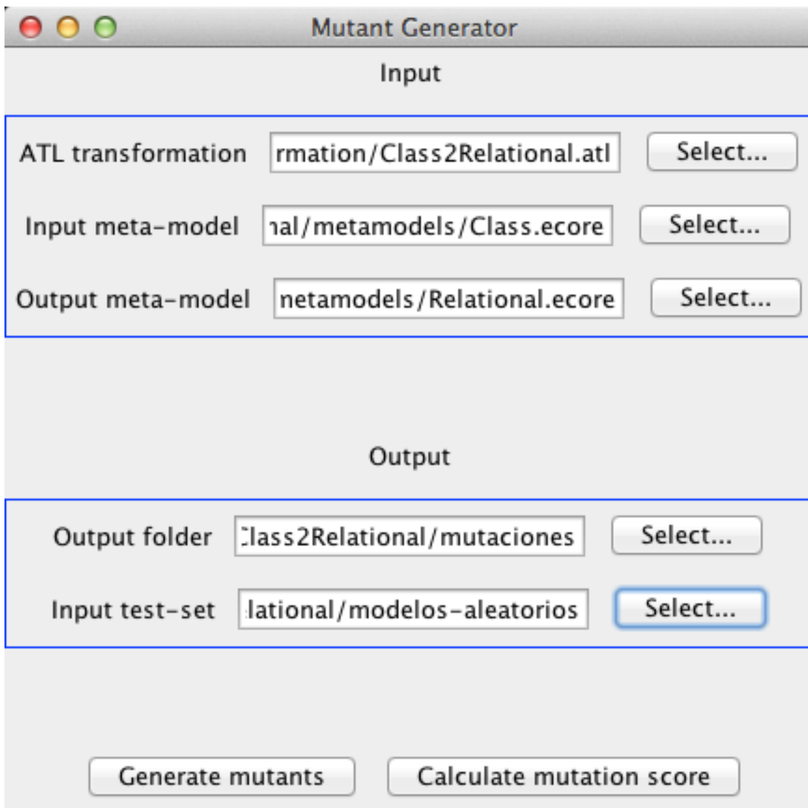

Figura 28. Pantalla de cálculo de porcentajes aleatorios

Y de nuevo generamos los porcentajes de mutación.

informeC2R-aleatorios.csv

	A	B	C	D	E	F
1		TOTAL	RSCC	ROCC	RSMA	RSMD
2	number of mutants	6.0	6	0	0	0
3	detected mutants	5.0	5	0	0	0
4	non-detected mutants	1	1	0	0	0
5	MUTATION SCORE	0.8333333	0.8333333	0.0	0.0	0.0
6						

Figura 29. Vista del fichero csv de salida para modelos aleatorios

Como podemos ver en la Figura 29, en este caso el numero de mutantes detectados es mayor, lo que el porcentaje de mutación también es mayor que en el caso anterior.

Con estos resultados podemos decir que el conjunto de casos de prueba generado de forma aleatoria es mejor que el conjunto de casos de prueba generado mediante las técnicas del documento [16]. Recordemos que las pruebas de mutación se usan para comprobar cómo de efectivo es un conjunto de casos de prueba; y para que un conjunto de casos de prueba sea eficaz, es necesario que localice el mayor numero de fallos posibles en una transformación. Por este motivo, como el primer conjunto de pruebas ha detectado un mayor número de mutantes podemos decir que será más efectivo para emplearlo a la hora de realizar las pruebas de esa transformación.

Hemos realizado el mismo experimento que se llevó a cabo en [16] utilizando una función oráculo distinta (en concreto, se usaba una especificación del comportamiento esperado de la transformación, en lugar de una comparación entre el modelo esperado y el modelo obtenido); en ese caso, el porcentaje de mutación salía peor para el conjunto de modelos aleatorios.

8 Conclusiones y trabajo futuro

El software de transformación de modelos no era algo con lo que hubiera trabajado anteriormente, por este motivo fue necesario realizar un estudio a fondo en este tema. Pero después de esto puedo decir que es un tema muy interesante, debido a que el Desarrollo de software Dirigido por Modelos plantea un enfoque más visual de forma que la implementación de programas utilice directamente los modelos de objetos que, en una implementación convencional utilizamos sólo para plantear el problema software que queremos abordar mediante código.

Tampoco conocía la existencia de las pruebas de mutación de software. Pero como se ha intentado demostrar en este trabajo, resultan de mucha utilidad. Cuando se están realizando pruebas de un cierto software muchas veces no sabemos si el conjunto de casos de prueba que estamos utilizando son suficientes o no; es posible que no estemos contemplando todos los casos o que los casos ya diseñados no cubran todas las características del programa probado.

Por este motivo, se considera de gran utilidad la herramienta que se ha propuesto en este trabajo de fin de grado. Una herramienta con la que podemos medir la calidad de un conjunto de pruebas para determinar si es necesario modificar dicho conjunto para incluir más casos. Si el conjunto de casos de prueba detecta todos los mutantes que se han creado podemos tener mayor certeza de que dicho conjunto proporciona una buena batería de pruebas de software.

Como trabajo futuro, queda el desarrollo del resto de tipos de mutaciones que se han propuesto en este documento, tales como las de filtrado y las de creación. De esta manera nuestra herramienta cubriría todos los casos de mutación, y esto proporcionaría un porcentaje de mutación mucho más preciso. De esta manera, las transformaciones para el lenguaje ATL tendrían disponible esta herramienta y podría ser usada por todos los usuarios que utilicen este lenguaje como método de programación de transformación de modelos.

9 Referencias

1 B. Boehm. 1981. Software Engineering Economics. Prentice-Hall.

2 Benoit Baudry, Trung Dinh-Trong, Jean-Marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, Yves Le Traon “Model Transformation Testing Challenges", In Proceedings of the IMDDMDT workshop at ECMDA’06, 2006.

3 J. M. Küster and M. Abd-El-Razik. "Validation of model transformations - First experiences using a white box approach". In MoDELS Workshops, volume 4364 of LNCS, pages 193-204. Springer, 2006.

4 C. A. González, J. Cabot: "ATLTest: A white-box test generation approach for atl transformations". In MoDELS'12, volume 7590 of LNCS, pages 449-464. Springer, 2012.

5 F. Büttner, M. Egea, J. Cabot: "On verifying ATL transformations using `off-the-shelf' SMT solvers". In MoDELS'12, volume 7590 of LNCS, pages 432-448. Springer, 2012.

6 F. Büttner, M. Egea, J. Cabot, M. Gogolla: "Verification of ATL transformations using transformation models and model finders". In ICFEM, volume 7635 of LNCS, pages 198-213. Springer, 2012.

7 J. Cabot, R. Clarisó, E. Guerra, J. de Lara: "Verification and validation of declarative model-to-model transformations through invariants". Journal of Systems and Software, 83(2):283-302, 2010.

8 T. A. Budd. 1981. "Mutation analysis: Ideas, examples, problems and prospects". Proc. of Computer Program Testing, pp. 129-148.

9 Jean-Marie Mottu, Benoit Baudry, Yves Le Traon: "Mutation Analysis Testing for Model Transformations". ECMDA-FA 2006, pp. 376-390.

10 Kermeta: http://www.kermeta.org/documents/user_doc

11 Tefkat: http://tefkat.sourceforge.net/tutorial1.html

12 Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev: "ATL: A model transformation tool". Sci. Comput. Program. 72(1-2): 31-39 (2008). Página web: http://www.eclipse.org/atl/

13 Eclipse: http://www.eclipse.org/

14 D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. 2008. EMF: The Eclipse Modeling Framework. 2nd Edition, Addison-Wesley Professional. http://www.eclipse.org/modeling/emf/

15 http://www.eclipse.org/emf/compare/index.html

16 "Specification-driven model transformation testing". 2013. Esther Guerra and Mathias Soeken. Software and Systems Modeling (Springer), best papers of ICMT'2012. In press.

17 Krzysztof Czarnecki, Simon Helsen: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3): 621-646 (2006).