UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

TRABAJO FIN DE MÁSTER

Un Framework para la Generación Automática de Ejercicios mediante Técnicas de Mutación

Máster Universitario en Investigación e Innovación en Tecnologías de la Información y las Comunicaciones (i^2-TIC)

Autor: Gómez Abajo, Pablo

Tutores: De Lara Jaramillo, Juan; Guerra Sánchez, Esther

Madrid, junio de 2016

Resumen / Abstract

En este trabajo se describe el diseño e implementación de un entorno que genera ejercicios tipo test de forma automática mediante técnicas de mutación, llamado Wodel-Edu. Wodel-Edu es una extensión de post-procesado para el Lenguaje de Dominio Específico (DSL) Wodel, desarrollado por el grupo MISO, y que proporciona primitivas de alto nivel para mutación de modelos. Para ello, se ha extendido el DSL Wodel con nuevas primitivas de mutación, nuevas estrategias de selección, un registro de las mutaciones aplicadas, un control de la generación de mutantes duplicados, y una comprobación de que los mutantes que se generan son modelos correctos (conformes a su meta-modelo). También se ha dotado a Wodel de un mecanismo extensible que permite registrar distintas acciones de post-procesado sobre los mutantes generados, extensión sobre la que se ha implementado el entorno Wodel-Edu. Wodel-Edu es independiente del dominio, y genera tres formatos diferentes de ejercicios tipo test: el primero, en el que se presentan varios diagramas, y el estudiante ha de decidir cuál es el correcto; el segundo, en el que se presenta un único diagrama, y el estudiante ha de decidir si es correcto, o no; el tercer formato, se presentan una serie de posibles cambios a realizar sobre el diagrama para corregirlo, y el estudiante ha de seleccionar cuáles de estos cambios son correctos. En este trabajo se ha elegido utilizar Wodel-Edu para generar ejercicios de autómatas finitos. Se presenta además una evaluación de la aplicación de ejercicios generada.

This work presents the design and development of a framework for the automatic generation of test exercises using mutation techniques, that we call Wodel-Edu. Wodel-Edu is a post-processing extension for the Domain-Specific Language (DSL) Wodel - developed by MISO group - that provides high level primitives for model mutation. We extend the DSL Wodel with new mutation primitives, new selection strategies, a registry of the applied mutations, a duplicated mutant generation control, and a verification that the generated mutants are conforming to the meta-model. We also improve Wodel with an extensible mechanism that allows applying post-processing actions to the generated mutants, and we use this feature to include the Wodel-Edu extension in the Wodel environment. Wodel-Edu is domain independent, and generates three kind of test exercises: the first one, where several diagrams are shown to the student, and he has to choose which one is correct; the second one, where just one diagram is shown to the student, and he has to choose if it is correct or not; and the third kind of exercise, where several changes, that can be applied to the diagram, are presented to the student, and he has to choose which of these changes are correct. In this work, we chose to apply Wodel-Edu to generate finite automata exercises. We also present an evaluation of the generated test application.

Palabras clave / Keywords

Lenguajes de Dominio Específico, Desarrollo Dirigido por Modelos, Mutación de Modelos, Educación, Corrección Automática de Ejercicios.

Domain-Specific Languages, Model-Driven Engineering, Model Mutation, Education, Automated Generation of Exercises.

Agradecimientos

Agradezco a Almudena, a mis padres, y a mis hermanos, que hayan estado siempre ahí, en las buenas y en las malas, y me hayan dado todo el cariño necesario para que yo haya podido llegar a este punto.

Agradezco a mis amigos, en especial a Isabel y a Víctor, su apoyo incondicional, y que cuenten conmigo para salir y despejarme un poco cuando lo necesito.

Agradezco a mis compañeros en los trabajos de las asignaturas del máster, Chema, José, Graciela, Gary, Lucrecia, Nancy, y Maximilian. Ha sido un placer trabajar con todos ellos.

Agradezco a mis profesores de las asignaturas del máster, ha sido mucho lo que he aprendido en este período, muy útil e interesante.

Agradezco a mis compañeros en el grupo MISO, ya que sin su ayuda no habría podido avanzar al mismo ritmo con el trabajo, y también a Víctor López Rivero, por su trabajo en el desarrollo original del DSL Wodel.

Agradezco a Alfonso Ortega de la Puente, mi primer tutor en el máster, que me dio la oportunidad de acceder a estos estudios.

Agradezco a mis tutores, Esther Guerra Sánchez y Juan de Lara Jaramillo, que apostasen por mí y me diesen la oportunidad de trabajar en el grupo MISO, y toda la labor que han realizado para que mi trabajo llegue a buen puerto.

Por último, agradezco a dos profesores míos de la carrera que después se han convertido en amigos, Josefina Sierra Santibáñez, y Manuel Alfonseca Moreno, su apoyo y compañía, y su orientación durante estos años.

Dedicado a Almudena

Contenido

Lista de Acrónimos

ACM	Association for Computing Machinery
ACME	Avaluació Continuada i Millora de l'Ensenyament (Evaluación Contínua y Mejora de la Enseñanza)
API	Application Programming Interface (Interfaz de Programación de Aplicaciones)
ATL	ATL Transformation Language (Lenguaje de Ttransformación ATL)
DSL	Domain-Specific Language (Lenguaje Específico del Dominio)
EMF	Eclipse Modeling Framework (Entorno de Modelado de Eclipse)
FTS	Feature Transition System (Sistema de Transiciones con Características)
IDE	Integrated Development Environment (Entorno Integrado de Desarrollo)
IEEE	Institute of Electrical and Electronics Engineers (Instituto de Ingenieros Eléctricos y Electrónicos)
LMS	Learning Management System (Sistema de Gestión del Aprendizaje)
MDE	Model-Driven Engineering (Ingeniería Dirigida por Modelos)
MISO	Modelling and Software Engineering Research Group, Escuela Politécnica Superior, Universidad Autónoma de Madrid
MOEA	Multiobjective Evolutionary Algorithms
MOF	Meta-Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
QVT	Query/View/Transformation (Conjunto estándar de lenguajes para transformación de modelos de la OMG)
SAC ‘16	31st ACM Symposium on Applied Computing
SQL	Structured Query Language
SUS	System Under Study (Sistema en estudio)
TLM	Transaction-Level Modeling (Modelado de Nivel de Transacción)
XML	eXtensible Markup Language (Lenguaje de Marcas Extensible)

Capítulo 1. Introducción

La Ingeniería Dirigida por Modelos (MDE) [7] es un paradigma de Ingeniería del Software que utiliza modelos como los principales referentes en todas las fases de desarrollo, y proporciona lenguajes específicos para su manipulación. Aunque existen muchos lenguajes de manipulación de modelos (por ejemplo, para transformación de modelos o generación de código), hay una falta de entornos de trabajo o lenguajes de dominio específicos para definir y aplicar mutación de modelos de manera general en múltiples dominios.

Un modelo mutante es una variación de un modelo original, creado por operaciones específicas de mutación de modelos, que dependerán del dominio de la aplicación. La mutación de modelos tiene muchas aplicaciones, por ejemplo, en áreas como pruebas de transformación de modelos [2], pruebas de software basadas en modelos [34], pruebas de programas [20], evaluación de algoritmos de detección de clones [30], generación de modelos grandes [22], generación automatizada de ejercicios [13, 25], o algoritmos genéticos [19].

Si bien existen muchas aplicaciones que utilizan mutaciones de modelos, en la práctica dichas mutaciones se han de especificar y programar de manera ad-hoc para cada dominio (p. ej., pruebas [2, 3]) y lenguajes concretos (p. ej., fórmulas lógicas [15]). Con el objetivo de facilitar la especificación y creación de mutaciones de modelos [27] para lenguajes y dominios arbitrarios, el grupo MISO[1] se planteó el desarrollo de un Lenguaje de Dominio Específico (DSL) llamado Wodel [13].

Wodel es un lenguaje que proporciona primitivas de alto nivel interesantes para mutación de modelos (p. ej., creación, borrado, inversión de referencias), estrategias de selección de elementos de un modelo (p. ej., aleatoria, específica, todos), y es posible construir mutaciones compuestas como secuencias de mutaciones simples, que posteriormente se traducen de forma automática a lenguaje Java.

En este trabajo se describe el diseño e implementación de Wodel, así como una aplicación a la educación de este lenguaje, que consiste en un entorno de generación automática de ejercicios tipo test que utiliza técnicas de mutación

[1] http://www.miso.es/

programadas en Wodel. Este entorno – Wodel-Edu – se desarrolla como plug-in para Eclipse [8, 39].

Una de las mayores ocupaciones de los docentes es la de elaborar ejercicios para los estudiantes, y corregirlos. El objetivo de Wodel-Edu es facilitar esta tarea. Para ello en este trabajo se ha ampliado el lenguaje Wodel con nuevas primitivas de mutación, (p. ej., inversión del valor de atributos booleanos, copia de atributos y referencias entre objetos, intercambio de atributos y referencias entre objetos), nuevas estrategias de selección (p. ej., selección de un objeto diferente a otro dado, selección de objetos que cumplen una serie de condiciones dadas), un registro del histórico de mutaciones aplicadas, un control de la generación de mutantes duplicados, y una comprobación de que los mutantes que se generan son modelos correctos (es decir, conformes a su meta-modelo).

También se ha dotado a Wodel de un mecanismo extensible que permite registrar distintas acciones de post-procesado sobre los mutantes generados. Esto facilita la integración de Wodel en otras aplicaciones que utilizan técnicas de mutación. Dicho mecanismo está disponible como un punto de extensión en Eclipse [29]. De este modo, Wodel-Edu implementa dicho punto de extensión, para permitir la generación de ejercicios a partir de los modelos generados mediante mutación.

Para la configuración de los ejercicios en Wodel-Edu se han implementado las extensiones de post-procesado con los siguientes DSLs: *modelDraw*, para la generación de ficheros gráficos a partir de los modelos; *eduTest*, para la configuración de la aplicación web de ejercicios; *idModel*, para definir las cadenas de texto con las que identificar a los elementos de un modelo; y *configOptions*, para configurar el texto de las opciones de respuesta que se presentan en los ejercicios de test de la aplicación web de ejercicios generados.

La utilización de técnicas de mutación para generación de ejercicios que se corrigen de forma automática consiste en la idea siguiente: dado un modelo de un determinado dominio, se pueden aplicar sucesivas mutaciones sobre este modelo. En cada paso en el que se aplican las mutaciones, éstas quedan registradas, teniendo el sistema toda la información necesaria: las mutaciones que se han aplicado en cada paso, y el modelo semilla sobre el que se han aplicado estas mutaciones. Contando con toda esta información, el sistema es capaz de identificar en cada modelo mutante cuáles son las instrucciones de mutación que se han aplicado, y de esta forma identificar cuáles corresponden al modelo mutante que se presenta en el ejercicio.

Wodel-Edu genera ejercicios de tres tipos diferentes. El primer y el segundo formato son ejercicios sencillos. El primer formato consiste en que la aplicación muestra al estudiante varios diagramas entre los que sólo uno es correcto; el estudiante ha de adivinar cuál es. En este tipo de ejercicios basta con presentar al estudiante un modelo semilla (sin mutaciones) entre otros modelos mutados. En el segundo formato de ejercicios, se muestra al estudiante un diagrama. El estudiante ha de indicar si es correcto o no. En este caso se presenta al estudiante un modelo elegido aleatoriamente, en caso de que sea un modelo semilla, el modelo es correcto, y en caso de que sea un modelo mutante, el modelo es incorrecto. El tercer formato de ejercicios es más complejo, consiste en presentar al estudiante un diagrama, que se

ha generado aplicando una o varias mutaciones sobre un modelo semilla. Se le muestran además varias opciones de texto a elegir, correctas e incorrectas, y el estudiante ha de descubrir cuáles son las correctas. En este tercer formato de ejercicios, las opciones correctas se generan a partir de las instrucciones de mutación aplicadas para generar el modelo mutante que se muestra en pantalla, y las opciones incorrectas se generan a partir de las instrucciones de mutación que se aplican para crear otros modelos mutantes que toman como semilla el modelo mutante presentado al estudiante. Wodel-Edu es capaz de corregir estos tres tipos de ejercicios de forma automática, identificando en cada caso las soluciones.

Wodel-Edu es independiente del dominio y, por tanto, puede configurarse para dominios diversos. En este trabajo se ha aplicado a generación de ejercicios de autómatas finitos. En este trabajo se presentan además los resultados de una evaluación realizada sobre los ejercicios generados.

Este trabajo con Wodel y Wodel-Edu ha dado lugar a un artículo [13] presentado en el congreso SAC ’16 celebrado en el pasado mes de abril en la ciudad de Pisa, con un índice de aceptación del 24,07%. A raíz de este congreso, hemos recibido una invitación para enviar una versión extendida del artículo a la revista “Computer Languages, Systems and Structures (Elsevier)”, que cuenta con un índice de impacto de 0,44.

1.1. Estructura de la memoria:

El resto del documento está organizado como sigue:

- Sección 2: Conceptos y técnicas de la Ingeniería Dirigida por Modelos utilizados en este trabajo.
- Sección 3: Estado del arte de utilización de técnicas de mutación en proyectos software, se hace una breve revisión sobre trabajos que utilizan técnicas de mutación realizados previamente.
- Sección 4: Descripción del DSL Wodel, arquitectura y herramienta.
- Sección 5: Wodel-Edu: Aplicación del DSL Wodel para la generación de ejercicios y evaluación de los mismos.
- Sección 6: Conclusión del trabajo y propuestas para trabajo futuro.

Capítulo 2. Ingeniería Dirigida por Modelos

La Ingeniería Dirigida por Modelos (MDE) utiliza el modelado como principal actividad del ciclo de vida de un sistema de software [7] (construcción, mantenimiento, ingeniería inversa, etc.). Este paradigma propone la construcción de modelos (abstracciones) de diferentes aspectos de un sistema y la transformación de dichos modelos de forma *semi*-automática. De esta forma, se consigue reducir errores en el proceso de Ingeniería del Software al aumentar el nivel de abstracción en la especificación del sistema y posibilitar la verificación y reutilización de los modelos y las transformaciones. Asimismo, se enfoca en aumentar la productividad reduciendo tiempos de desarrollo a través de mecanismos automáticos de construcción.

En esta sección se hace un breve repaso sobre las distintas técnicas, y conceptos, de MDE utilizados en este trabajo, tanto en el diseño e implementación del DSL Wodel como en el de su extensión Wodel-Edu.

2.1. Modelos y Meta-modelos

En el contexto de MDE, se define un sistema "como un concepto genérico para el diseño de una aplicación software, una plataforma software, o cualquier otro artefacto software" [24]. Además, un sistema puede estar compuesto de otros subsistemas, así como tener relaciones con otros sistemas (p. ej., un sistema se puede comunicar con otros).

En base a esta definición de sistema, un modelo es una abstracción de un sistema en estudio (SUS, *system under study*), que existe en este momento o se pretende que exista en un futuro. Un modelo ayuda a definir un sistema y dar respuestas sobre el sistema estudiado sin la necesidad de considerarlo directamente.

En el ámbito de la Ingeniería del Software, la utilización de MDE permite organizar la información de una forma estructurada, lo que facilita la comprensión del proceso que se representa. La MDE tiene como efecto inmediato que el ingeniero de software puede centrarse en la lógica del sistema que está creando, liberándose de aquellas tareas más mecánicas y repetitivas del desarrollo software, que mediante estas técnicas quedan en gran medida automatizadas.

Cualquier proceso MDE debe tener un objetivo, p. ej., definición de requisitos, diseño/desarrollo de una aplicación, análisis y pruebas de sistemas, etc. Este objetivo es el que se define en la sintaxis abstracta del modelo. Un meta-modelo describe esta sintaxis abstracta del modelo, y se dice que los modelos son conformes a este meta-modelo.

Para definir un meta-modelo, primero es necesario definir lo que es un lenguaje de modelado. Un lenguaje de modelado es un lenguage artificial que puede utilizarse para expresar una información, un conocimiento, o un sistema, en una estructura que queda definida por un conjunto de reglas coherentes. Estas reglas se utilizan para interpretar el papel de los elementos dentro de esta estructura [14].

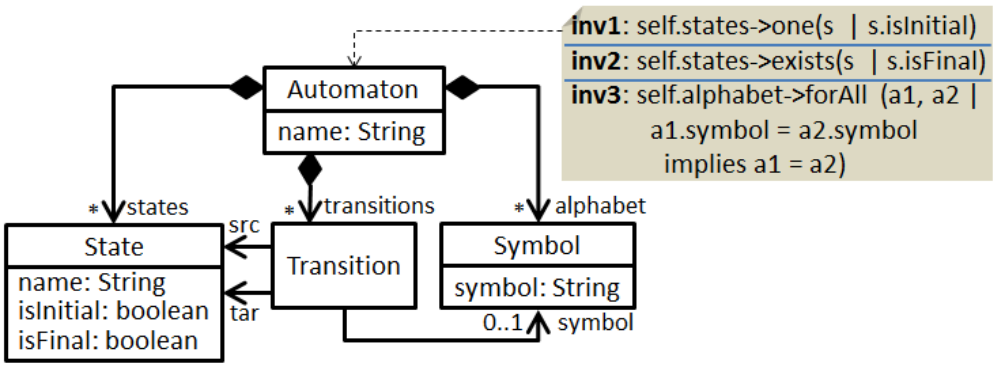

Figura 2.1 Meta-modelo para autómata finito

Un meta-modelo es una representación que describe los elementos que pueden utilizarse en un lenguaje de modelado, y cómo se relacionan entre sí. La técnica del meta-modelado consiste en la creación de modelos a partir de un lenguaje de modelado. El meta-modelo es la representación abstracta de las características que tiene este lenguaje. Por ejemplo, el Unified Modeling Language (UML) [44] establece que dentro de un modelo se pueden utilizar los elementos Clase, Atributo, Asociación, Paquete, etc. La Figura 2.1 corresponde a un meta-modelo para automátas finitos. Un meta-modelo describe la *sintaxis abstracta* del lenguaje: los conceptos del lenguaje y las relaciones entre ellos, así como las reglas que indican cuándo un modelo está bien formado.

Cuando es necesario incluir en un meta-modelo reglas que no se pueden expresar únicamente en términos de clases y relaciones, se utiliza el Object Constraint Language (OCL) [42], que es un lenguaje declarativo para describir reglas que se aplican a modelos UML. Aunque inicialmente OCL era únicamente una extensión formal a la especificación del lenguaje UML, en la actualidad forma parte de este estándar. OCL puede utilizarse con cualquier meta-modelo MOF, incluyendo UML [36]. En la Figura 2.1 se indican tres restricciones OCL: *inv1* indica que en el modelo debe existir un único estado inicial; *inv2* indica que en el modelo al menos debe haber un estado final; e *inv3* indica que los símbolos del alfabeto han de ser diferentes.

Un modelo es una instancia de un meta-modelo, y se dice que es conforme a este meta-modelo si: los elementos del modelo son instancias de las clases del meta-modelo; las relaciones en el modelo son instancias de las asociaciones en el meta-

modelo; se satisface la cardinalidad en las asociaciones, las claves únicas, y las restricciones OCL incluidas en el meta-modelo. La Figura 2.2 muestra un modelo para autómatas finitos que aceptan números binarios pares, y que es conforme al meta-modelo de la Figura 2.1.

La sintaxis abstracta consiste únicamente en la estructura de los datos, a diferencia de la *sintaxis concreta*, que incluye además información sobre la representación. Por ejemplo, una sintaxis concreta puede incluir características como los paréntesis (para agrupar), o comas (para listas), que no aparecen en la sintaxis abstracta, ya que van implícitos en la estructura. Por tanto, la sintaxis concreta sirve para describir la notación del lenguaje de modelado. Puede ser gráfica, como se observa en la Figura 2.3, o textual, como se observa en la Figura 2.4.

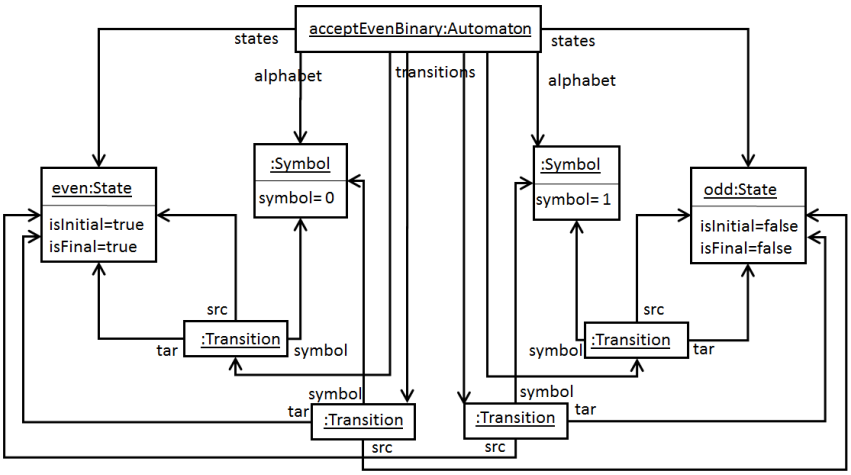

Figura 2.2 Modelo de un autómata finito que acepta números binarios pares

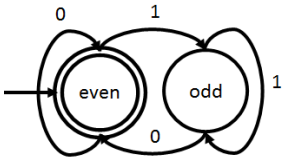

Figura 2.3 Sintaxis concreta gráfica para un autómata finito que acepta números binarios pares

```
Automaton acceptEvenBinary {
    states {
        even(isInitial, isFinal),
        odd
    }
    transitions {
        src even tar even symbol '0',
        src even tar odd symbol '1',
        src odd tar even symbol '0',
        src odd tar odd symbol '1'
    }
    alphabet {
        '0',
        '1'
    }
}
```

Figura 2.4 Sintaxis concreta textual para un autómata finito que acepta números binarios pares

2.2. Lenguajes de Dominio Específico

En desarrollo de software e ingeniería de dominio, un Lenguaje de Dominio Específico (DSL) es un lenguaje de programación o especificación dedicado a resolver un problema en particular, representar un problema específico, y proporcionar una técnica para solucionar una situación particular [18]. El concepto no es nuevo, pero se ha vuelto más popular debido al aumento del uso de modelado específico del dominio. Con la implementación de un DSL no se pretende proporcionar características para resolver cualquier tipo de problema, y seguramente con un DSL no se pueden implementar todos los programas que se pueden hacer con, p. ej., Java o C (que se denominan Lenguajes de Propósito General) [4].

Los DSLs están adquiriendo cada vez mayor importancia en la Ingeniería del Software. Las herramientas son también cada vez mejores, de forma que se puede desarrollar un DSL con relativamente poco esfuerzo [35]. Estas herramientas permiten diseñar los DSLs de forma gráfica o textual.

Algunos ejemplos de DSL son: R[2] para estadísticas, Mata[3] para programación matricial, Mathematica[4] para matemáticas, fórmulas de hojas de cálculo y macros, SQL [26] para consultas a bases de datos relacionales, etc.

Crear un DSL (con software que lo soporte) adquiere todo el sentido cuando permite que pueda expresarse un tipo particular de problemas o soluciones más claramente que con otros lenguajes existentes, y el tipo de problema se presenta de forma suficiente. En los DSL, la semántica del lenguaje está muy cercana al dominio del problema para el cual se diseña. Tienen un alto nivel de abstracción al usuario, por tanto, están dirigidos a expertos en el dominio.

A continuación, se describe el framework Xtext que se ha usado para definir los DSLs que se han creado en este trabajo, y que se presentan en las secciones 4 y 5.

2.3. Eclipse Modeling Framework

El Eclipse Modeling Framework (EMF) [29] es un entorno de modelado que aprovecha las facilidades proporcionadas por Eclipse para definir un modelo en cualquiera de estas tres formas (interfaces Java, diagramas UML, o esquemas XML), a partir de las cuales se pueden generar las otras dos, y también la correspondiente implementación de las clases.

EMF está integrado y mejorado para una programación eficiente. Responde a la pregunta que uno se hace habitualmente: "¿Debo modelar o debo programar?", con esta respuesta: "Puedes elegir cualquiera de las dos".

---

[2] https://www.r-project.org/

[3] http://www.stata.com/

[4] http://www.wolfram.com/mathematica/

Con EMF, el modelado y la programación se pueden considerar como la misma cosa. En lugar de forzar una separación entre la ingeniería de alto nivel del modelado, y la programación de bajo nivel, consigue que sean dos partes bien integradas del mismo trabajo. Muchas veces, especialmente con aplicaciones grandes, este tipo de separación todavía es deseable, pero con EMF uno mismo decide el grado en el que se da esta separación.

El formato utilizado para representar meta-modelos en EMF es Ecore, que es a su vez un modelo EMF. Ecore es un subconjunto simplificado de UML. Ecore permite definir los elementos de un meta-modelo mediante clases Ecore, (p. ej., *EClass*, para representar la clase modelada, con un nombre, cero o más atributos, y cero o más referencias; *EAttribute*, para representar el atributo modelado, que tiene un nombre y un tipo; *EReference*, para representar un extremo de una asociación entre clases, con un nombre, un valor booleano que indica si es una referencia contenedora, y el tipo de la referencia destino; etc.).

Un proyecto EMF de Eclipse genera de forma automática las clases Java necesarias para manejar los modelos conformes al meta-modelo Ecore, y se puede añadir la funcionalidad adicional que se desee. EMF proporciona una API [29] mediante la que se manejan los objetos EMF de forma genérica. Utiliza la representación canónica XMI (XML Metadata Interchange) para la serialización directa de los modelos Ecore y sus instancias, de forma que no se almacena ninguna información extra.

EMF es el estándar “de facto” de modelado y una implementación de MetaObject Facitlity (MOF) de la OMG [41].

2.4. Xtext y Xtend

Xtext [4] es un framework de Eclipse de código abierto para implementar DSLs textuales e integrarlos con el IDE de Eclipse. Xtext permite implementar lenguajes rápidamente, cubriendo todos los aspectos de un entorno completo de lenguaje: analizador sintáctico, generador de código, o intérprete, alcanzando una integración completa con el IDE de Eclipse (con todas las características típicas de un IDE como un editor con coloreado de sintaxis, completado de código, marcadores de errores, infraestructura automática de compilado, etc.).

Xtend [4] es un lenguaje de programación parecido a Java capaz de interactuar completamente con el sistema de tipos de Java. Tiene una sintaxis sencilla y compacta, y características avanzadas como inferencia de tipos o expresiones lambda. Xtext fomenta la utilización de Xtend, siendo Xtend el lenguaje empleado en las clases que se generan automáticamente al crear un proyecto Xtext en Eclipse. Xtext está implementado a partir de EMF, lo que conlleva que cualquier programa escrito en el código de un DSL creado con Xtext es un modelo de EMF, y puede ser serializado como modelo XMI.

2.5. Transformación de Modelos

Además de los modelos, la transformación de modelos representa el otro ingrediente esencial de MDE, y permite la definición de mapeados entre modelos diferentes. Las transformaciones se definen en el nivel del meta-modelo, y se aplican en el nivel del modelo. La transformación se realiza entre un modelo de entrada y uno de salida, pero se define realmente en los respectivos meta-modelos [7] (ver Figura 2.5).

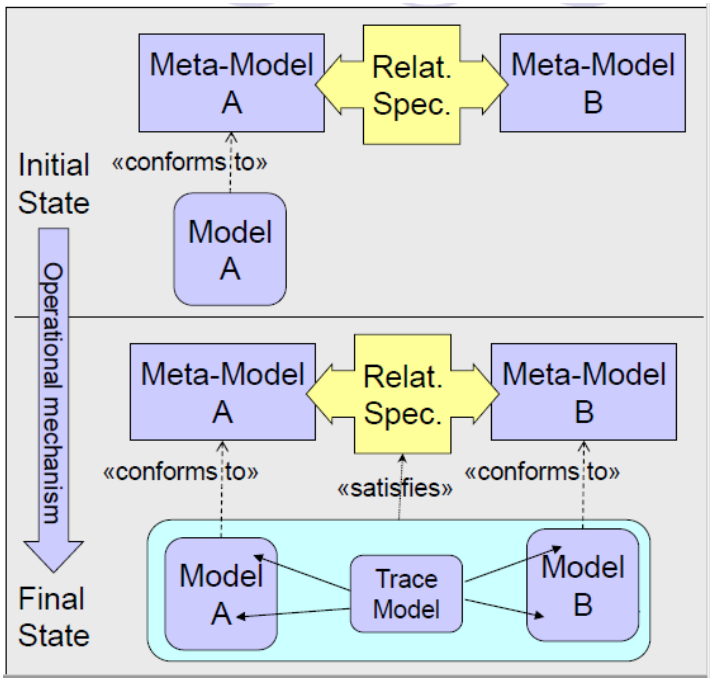

Figura 2.5 Transformación de modelos *out-place*

La transformación de modelos sirve para realizar análisis de sistemas, optimización de código, rediseño, refactorización, simulación, etc. Esta transformación de modelos puede ser de distintos tipos: *in-place*, en la que los cambios se aplican dentro del propio modelo de entrada; o *out-place*, en la que se convierte un modelo de entrada conforme a un meta-modelo, a un modelo de salida conforme a otro meta-modelo.

MDE proporciona lenguajes apropiados para definir transformaciones de modelos, y también proporciona a los diseñadores soluciones optimizadas para especificar reglas de transformación. Estos lenguajes pueden utilizarse para definir transformaciones de modelos en términos de plantillas de transformación que se aplican normalmente sobre los modelos de acuerdo a reglas que se comprueban en los elementos de los modelos.

Algunos lenguajes relevantes de transformación de modelos son: QVT [43] y ATL [38], para transformación de modelos *out-place*, Henshin [40], para transformación de grafos [11], etc.

La mutación de modelos es una técnica que está muy relacionada con la transformación de modelos. Al igual que en la transformación de modelos, hay uno o

varios modelos de entrada, que en mutación de modelos son los modelos 'semilla', y hay uno o varios modelos de salida, que en mutación de modelos son los modelos 'mutantes'. Por tanto, la mutación de modelos es un caso especial de transformación de modelos, en la que los modelos de entrada y los modelos de salida son conformes al mismo meta-modelo, y en la que se especifican los cambios o 'mutaciones' que se van a introducir en los modelos. En este trabajo se ha implementado un registro de mutaciones (ver Sección 4.4), que es la traza de los cambios entre el modelo semilla y los modelos mutantes generados. En la Figura 2.6 se muestra un esquema de la mutación de modelos.

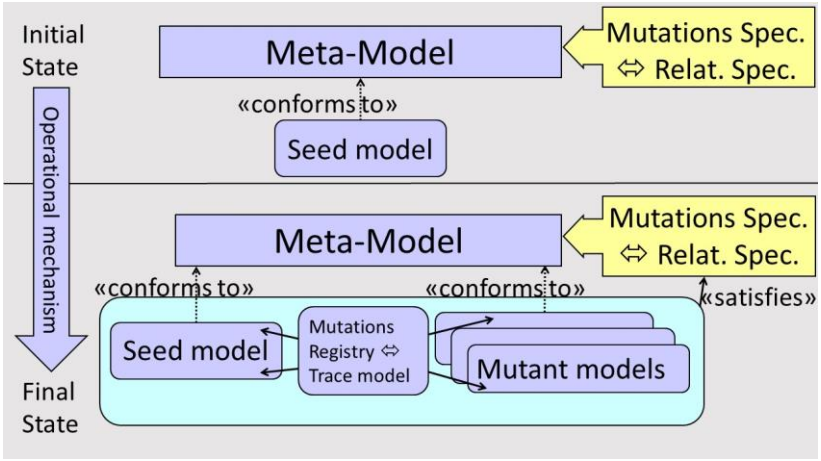

Figura 2.6 Mutación de modelos

2.6. Generación de Código

La generación de código es un tipo especial de transformación de modelos que tiene como objetivo producir código de un nivel más bajo a partir de un modelo para crear una aplicación que funcione, de forma muy similar a la que utilizan los compiladores para producir ficheros binarios ejecutables partiendo de código fuente. En este sentido, los generadores de código son a veces denominados *compiladores de modelos* [7].

Esta generación se hace normalmente por medio de un motor de plantillas basadas en reglas (ver Figura 2.7), p. ej., un generador de código que consiste en un conjunto de plantillas con controles de contenido que en cuanto se aplican (se instancian) sobre los elementos del modelo, producen el código. En este trabajo se ha utilizado el lenguaje de plantillas Xtend, proporcionado por Xtext, para generar el código Java a partir de las instrucciones de mutación en el lenguaje Wodel, y el código html a partir de las configuraciones en el plug-in Wodel-Edu.

Una vez el código es generado, pueden emplearse las herramientas habituales del IDE con el que se esté trabajando para optimizar el código fuente producido durante la generación (si es necesario), compilarlo, y finalmente desplegarlo.

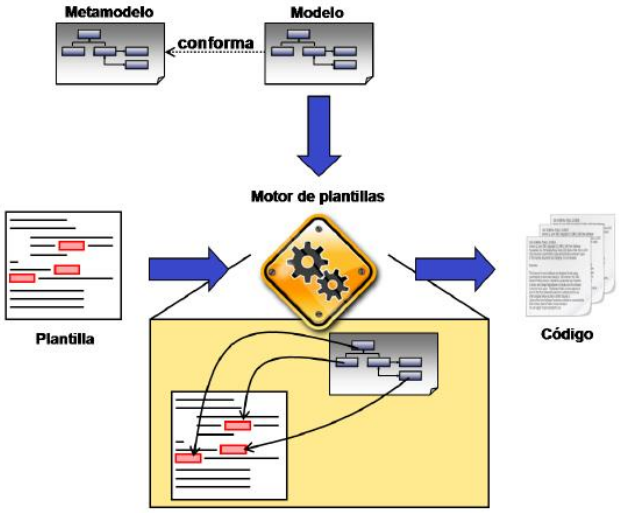

Figura 2.7 Arquitectura de un generador de código

Capítulo 3. Estado del arte

La revisión de otros trabajos relacionados se divide en dos partes. Por un lado, se revisan trabajos recientes que han utilizado técnicas de mutación con diferentes objetivos, encontrando un trabajo que utiliza técnicas de mutación para generación de ejercicios [25]. Por otro lado, se revisan trabajos relacionados con la generación automática de ejercicios [23, 28], sin tener en cuenta las técnicas utilizadas.

3.1. Estado del arte de técnicas de mutación

En este apartado se hace una revisión de otros trabajos que han utilizado técnicas de mutación. Se observa que estas técnicas se utilizan en campos diversos, como transformación de modelos [2], sistemas adaptativos [3], pruebas de software basadas en modelos [34], pruebas de programas [1, 16, 20, 27], fórmulas lógicas [15], generación de modelos grandes [22], algoritmos genéticos [19], generación de ejercicios [25], algoritmos de detección de clones [30], pruebas de sistemas de transiciones [10], y pruebas de protocolos de comunicaciones [6].

3.1.1. Criterio utilizado

Se ha realizado un Systematic Mapping Study (método para construir una clasificación y estructurar un campo de interés de Ingeniería del Software) en la ACM Digital Library[5] y en IEEE Xplore Digital Library[6], utilizando la siguiente consulta, y seleccionando únicamente artículos desde el 2010:

'mutation analysis model' o

'mutation testing model' o

'mutation language model'

[5] http://dl.acm.org/

[6] http://ieeexplore.ieee.org/

En el primer filtro, se seleccionan sólo aquellos artículos relacionados con Ingeniería del Software, ya que se encuentran bastantes artículos relacionados con la biología, u otras áreas. Como resultado se obtienen 41 artículos.

En el segundo filtro se seleccionan aquellos artículos que están más relacionados con nuestro tema de estudio: técnicas de mutación en Ingeniería del Software. Este filtro consiste en seleccionar minuciosamente aquellos artículos que han resultado de mayor interés, por estar más relacionados con el tema de investigación. Como resultado se obtienen 19 artículos.

Por último, tras revisar estos artículos, se realiza el filtro final, en el que se seleccionan los más relevantes con respecto a este trabajo. Como resultado se obtienen 4 artículos. La Tabla 3.1 resume los resultados de este estudio.

	Artículos obtenidos (desde el 2010)	Primer filtro (palabras clave)	Segundo filtro (Ingeniería del Software)	Filtro final (+relevantes)
ACM DL	45	25	10	3
IEEE Xplore DL	37	16	9	1
Total	82	41	19	4

Tabla 3.1 Resultados de la búsqueda sistemática de artículos relacionados con técnicas de mutación

Además de estos 4 artículos, se revisan también los 10 artículos referenciados en el artículo 'Wodel: A Domain-Specific Language for Model Mutation' [13], presentado recientemente en el congreso SAC '16 de la ACM. Por tanto, el total de artículos utilizados en este trabajo es de 14.

3.1.2. Herramientas, lenguajes y frameworks

En esta sección se revisan herramientas que utilizan técnicas de mutación, lenguajes para la definición de mutaciones, y entornos disponibles para aplicar técnicas de mutación. Esta es la categoría en la que más trabajos se han encontrado, un total de 12. Estos trabajos corresponden a áreas como la transformación de modelos [2], sistemas adaptativos [3], pruebas de software basadas en modelos [34], pruebas de programas [1, 16, 27], fórmulas lógicas [15], generación de modelos grandes [22], generación de ejercicios [25], algoritmos de detección de clones [30], optimización de algoritmos genéticos [19], y pruebas de sistemas de transiciones [10].

Arenaga et al. [2] presentan una herramienta para mejorar la calidad de un conjunto de datos de prueba. En su trabajo, que corresponde al dominio de la transformación de modelos, permiten producir automáticamente o semi-automáticamente modelos de prueba mejorados a partir de un conjunto de modelos de prueba existentes, utilizando para ello operadores de mutación predefinidos dependientes de la transformación, trazas de la ejecución de la transformación, y un algoritmo dedicado.

Bartel et al. [3] proponen una herramienta para realizar pruebas de sistemas adaptativos mediante técnicas de mutación dirigidas por modelos. Utilizan tres operadores de mutación en este dominio de los sistemas adaptativos: ignorar la propiedad del contexto, ignorar el valor de la propiedad del contexto específico, e ignorar los valores de la propiedad de múltiples contextos.

Simão y Maldonado [27] proponen el lenguaje Mudel, y un entorno para introducir mutaciones en el código de programas para realizar pruebas de mutación. Para ello utilizan los lenguajes Yacc y Lex[7]. Las mutaciones realizadas son independientes del contexto.

Henard et al. [15] desarrollan la herramienta Mutalog para mutación de fórmulas lógicas. Las fórmulas sobre las que se aplican las mutaciones están en Forma Normal Conjuntiva (CNF). Mutalog está implementado en el lenguaje Java e incluye las siguientes operaciones de mutación: omisión o negación de un literal, omisión o negación de una cláusula, transformación de OR en AND y viceversa. El objetivo de este trabajo es evaluar la calidad de los entornos de pruebas.

Pietsch et al. [22] describen SiDiff, un sistema generador de modelos con propiedades predefinidas. Permite crear modelos, modificar los modelos existentes, y se crean históricos de modelos. Los modelos son conformes al meta-modelo, y también a las restricciones adicionales (normalmente expresiones OCL). Permite dos interpretaciones para las mutaciones: la interpretación literal, en la que se efectúan las operaciones el número de veces que se indica, y la interpretación estocástica, en la que se efectúan las operaciones en base a una frecuencia que se interpreta como probabilidad. Las estrategias de selección que utilizan provienen de algoritmos genéticos. SiDiff proporciona operadores de mutación de creación, pero no de borrado o modificación de modelos existentes, que podrían ser útiles para ciertas aplicaciones.

Sadigh et al. [25] proponen un sistema para la generación automática de ejercicios. Los problemas han de estar basados en modelos (problemas de máquinas de estados) y constan de los propios modelos, las propiedades (que el modelo debe satisfacer), y las trazas (indicadores de que el modelo tiene el comportamiento deseado o no).

Stephan et al. [30] utilizan técnicas de mutación para comparar frameworks de detección de clones de modelos. En su trabajo describen tres tipos de clones de modelos: los clones exactos, en los que se ignoran las variaciones en la presentación visual; los clones renombrados, en los que se ignoran las variaciones en las etiquetas, valores, tipos y las variaciones de clones exactos; y los clones con cambios de posición o de conexión, y pequeñas adiciones o eliminaciones de bloques y líneas, además de los cambios de clones exactos y de clones renombrados.

Vincenzi et al. [34] presentan Muta-Pro, con el que definen un proceso para aplicar pruebas de mutación de forma incremental. Su planteamiento parte de que el coste y la eficacia de las pruebas de mutación están relacionados con el coste y la

[7] http://dinosaur.compilertools.net/

eficacia del conjunto de operadores de mutación. Su propuesta optimiza estas cualidades.

Aicherning et al. [1] describen la generación de casos de prueba basados en modelos a partir de máquinas de estado UML, y el análisis de mutación en los modelos para generar casos de prueba. Utilizan un sistema de alarma para coches como prueba de concepto.

Lackner y Schmidt [16] desarrollan un sistema de mutación para la estimación de la calidad de pruebas de líneas de productos software. Definen operadores de mutación para modelos de características, máquinas de estado UML, y modelos de mapeado. Describen tres casos de estudio: una tienda online, una máquina expendedora de billetes, y un sistema de alarma.

Moawad et al. [19] describen la aplicación de técnicas de diseño de MDE para definir las funciones de fitness y los operadores de mutación en los algoritmos genéticos multiobjetivo (MOEAs), de forma que no es necesario tener conocimiento de la codificación MOEA. Explican que los problemas resueltos mediante MOEAs ad-hoc requieren conocimientos amplios del dominio del problema por parte de los técnicos que desarrollan la solución. Proponen que se pueden aprovechar las técnicas de MDE para que los desarrolladores sólo tengan que preocuparse de la lógica para resolver el problema, independientemente del dominio del mismo. Indican tres operadores de mutación: AddInstance, RemoveInstance, y ChangeWeight. Como caso de estudio, muestran una aplicación de su trabajo en un problema de distribución de tareas de una serie de aplicaciones en la nube. Una vez evaluado el caso de estudio demuestran que la solución aportada que utiliza técnicas MDE tiene un rendimiento similar a las soluciones existentes ad-hoc.

Devroey et al. [10] presentan su Framework Variability-Intensive Behavioural teSting (VIBeS). Representan todos los posibles mutantes de sistemas de transiciones con características (FTSs) (un FTS consiste en una representación formal y compacta del comportamiento de una línea de producto software [9]) en un único modelo restringido por un modelo de características para activación o desactivación de mutantes. Esto permite evaluar todos los mutantes del FTS en una única ejecución de un caso de prueba.

3.1.3. Metodologías

Nguyen et al. [20] describen una metodología para realizar pruebas de delegación de permisos entre los usuarios mediante técnicas de mutación. Plantean la problemática de probar de forma rigurosa la delegación de permisos entre los usuarios. Para ello se propone el uso de técnicas de mutación para cambiar los permisos y roles de diferentes usuarios. Como prueba de concepto se realizan pruebas de mutación en un sistema gestor de bibliotecas (Library Management System).

Bombieri et al. [6] presentan una metodología para aplicar análisis de mutación a la medición de la calidad del banco de pruebas de verificación de los protocolos Transaction-Level Modeling (TLM). La calidad de los bancos de pruebas considera la

corrección del protocolo TLM y la conformidad con un estándar predefinido (p. ej., OSCI TLM), una optimización del tiempo de simulación durante el análisis de mutación evitando redundancias, y una orientación de los diseñadores a la mejora del banco de pruebas.

3.2. Estado del arte de generación automática de ejercicios

A continuación, se hace una revisión de trabajos de generación y evaluación automática de ejercicios, encontrando principalmente trabajos en evaluación de ejercicios de programación. Algunos sitios web de evaluación automática de ejercicios de programación son, por mencionar algunos, las páginas de *CodinGame*[8] y *Acepta el reto*[9].

Se ha mencionado en la Sección 3.1.2 el trabajo de Sadigh et al. [25], que genera ejercicios de máquinas de estado de forma automática utilizando técnicas de mutación. Queirós y Leal [23] proponen la herramienta *Petcha* (*Programming Exercises TeaCHing Assistant*). Petcha es una herramienta de Asistencia Automática a la Enseñanza en cursos de programación. El objetivo final de Petcha es incrementar el número de ejercicios de programación resueltos satisfactoriamente por los estudiantes. Petcha sirve de asistente tanto para los profesores a la hora de crear los ejercicios como para los estudiantes para resolver estos ejercicios.

En el artículo de Queirós y Leal se hace una revisión de varios sistemas de creación automática de ejercicios de programación, y también de otros sistemas de evaluación automática de ejercicios de este tipo. Se describe que, a pesar de algunos intentos [12, 33] para definir un formato común de descripción de ejercicios de programación, cada uno de estos sistemas tiene su propio formato.

Uno de estos sistemas es *JExercise* [31], un plug-in para Eclipse basado en especificación, y orientado a pruebas. *JExercise* consiste en una especificación de los elementos requeridos y el comportamiento deseado, un conjunto de pruebas JUnit para verificar el código, y un modelo de la solución requerida.

En cuanto a los sistemas de evaluación automática de ejercicios, se indica que muchos de ellos proporcionan otras características, como soporte multi-lenguaje de programación, tipo de evaluación estática o dinámica, feedback, interoperabilidad, contexto de aprendizaje, seguridad, y plagio. El enfoque habitual de este tipo de evaluación es el de compilar el código y ejecutar el programa con un conjunto de casos de prueba con ficheros de entrada y salida (enfoque de caja negra). El programa se cataloga como aceptado si compila sin errores, y la salida de la ejecución de cada prueba es la misma que la de la salida esperada.

Otros sistemas [5, 17] no sólo prueban el comportamiento de programas sueltos, sino que analizan la estructura del código fuente (enfoque de caja blanca).

---

[8] https://www.codingame.com/

[9] https://www.aceptaelreto.com/

Petcha permite a los profesores: crear ejercicios de programación, desplegar los ejercicios en un repositorio, y configurar la actividad de programación en un LMS (Learning Management System). Petcha permite a los estudiantes: seleccionar una actividad en el LMS, y ejecutar la actividad utilizando el IDE y Petcha.

Soler et al. [28] presentan una herramienta de e-learning orientada a web para diagramas de clases UML. El entorno UML que proponen es capaz de corregir automáticamente ejercicios de diagramas de clases UML y proporcionar un feedback al estudiante de forma inmediata. La herramienta forma parte de un entorno más general, llamado ACME [37], que proporciona las funcionalidades principales de una plataforma de e-learning.

El estudiante dibuja el diagrama UML en la interfaz gráfica de la herramienta, que permite que se introduzcan diagramas hasta que se dé con la solución o se llegue al límite de tiempo. El sistema registra todos los intentos hasta que se obtiene la solución. Toda esta información es muy valiosa para el profesor, y puede utilizarse para la evaluación del alumno.

Para cada ejercicio se almacena un conjunto de posibles soluciones, que se codifican utilizando un formato específico. Una vez el estudiante introduce su solución, se procede a corregirlo mediante un proceso de comparación. Se compara la solución introducida por el estudiante con todas las soluciones posibles del problema almacenadas en el repositorio del sistema. Si ninguna de las soluciones correctas coincide con la del estudiante, el sistema marca la solución del estudiante como incorrecta, y le envía un mensaje de feedback utilizando la solución que más se acerca a la que él ha enviado, orientándole de esta forma a llegar a la solución.

3.3. Conclusiones del estudio

Del estudio del arte realizado sobre técnicas de mutación, se puede concluir que existen algunos frameworks y herramientas para mutación de modelos, pero son específicos para un lenguaje (p. ej., fórmulas lógicas [15]), o para un dominio (p. ej., pruebas [2, 3]); además, los operadores de mutación se crean normalmente utilizando lenguajes de programación de propósito general, que no están orientados a la definición y generación de mutantes [13]. Esto implica un mayor trabajo para el programador, que debe enfrentarse a detalles accidentales de la plataforma de implementación, por ejemplo, para realizar la carga y serialización de modelos, comprobar si los mutantes generados son válidos, etc.

Por tanto, existe una carencia de propuestas que faciliten la definición de operadores de mutación, que pueda aplicarse a diferentes lenguajes y aplicaciones. Esto facilitaría la creación de frameworks de mutación específicos del dominio como los mencionados en la Sección 3.1, proporcionando: primitivas de alto nivel (p. ej., para la creación de objetos o la redirección de referencias) a la vez que estrategias para su composición; integración con aplicaciones externas a través de la compilación a un lenguaje de propósito general; trazabilidad de las mutaciones aplicadas; y comprobación de mutantes repetidos.

El DSL Wodel presentado en este trabajo es la propuesta del grupo MISO para facilitar estas tareas al utilizar técnicas de mutación en proyectos software.

Por otro lado, los entornos de generación automática de ejercicios revisados también son específicos para un dominio determinado, como ejercicios de programación [23], o ejercicios de diagramas de clases UML [28]. Por tanto, esta propuesta del entorno Wodel-Edu de generación automática de ejercicios independiente del domino es un enfoque novedoso.

Capítulo 4. Wodel: Un DSL para la creación de operadores de mutación de modelos

Este capítulo introduce el lenguaje Wodel, que es un DSL para la definición y aplicación de mutaciones en modelos. En primer lugar, la Sección 4.1 presenta la arquitectura del entorno que se ha desarrollado como plug-in de Eclipse para el DSL Wodel. En segundo lugar, la Sección 4.2 presenta características del DSL Wodel, como las primitivas de mutación que ofrece, y la expresividad y concisión de este DSL. La Sección 4.3 presenta la extensión al lenguaje Wodel para registrar las mutaciones aplicadas.

4.1. Arquitectura

En MDE, los modelos deben ser conformes a un meta-modelo que define los elementos, propiedades y relaciones admisibles en el modelo. El objetivo es, por tanto, proporcionar un DSL para especificar operadores de mutación, la estrategia con la que se aplican a modelos conformes a un meta-modelo arbitrario dado, y facilitar la utilización de los mutantes generados para diferentes aplicaciones.

La Figura 4.1 muestra el flujo de trabajo del entorno propuesto. Primero el usuario proporciona un conjunto de modelos semilla conformes a un meta-modelo (etiqueta 1). A continuación, el usuario utiliza Wodel para definir los operadores de mutación deseados y sus detalles de ejecución, como cuántas mutaciones de cada tipo deben aplicarse en cada mutante, o su orden de ejecución (etiqueta 2). Además, cada programa Wodel necesita declarar el meta-modelo de los modelos que se van a mutar, que puede ser cualquiera, ya que Wodel es independiente del meta-modelo. De esta forma se posibilita la comprobación de tipos en el programa para asegurar que sólo se hace referencia a tipos y propiedades válidos del meta-modelo, y permite también comprobar que el resultado de la mutación es válido.

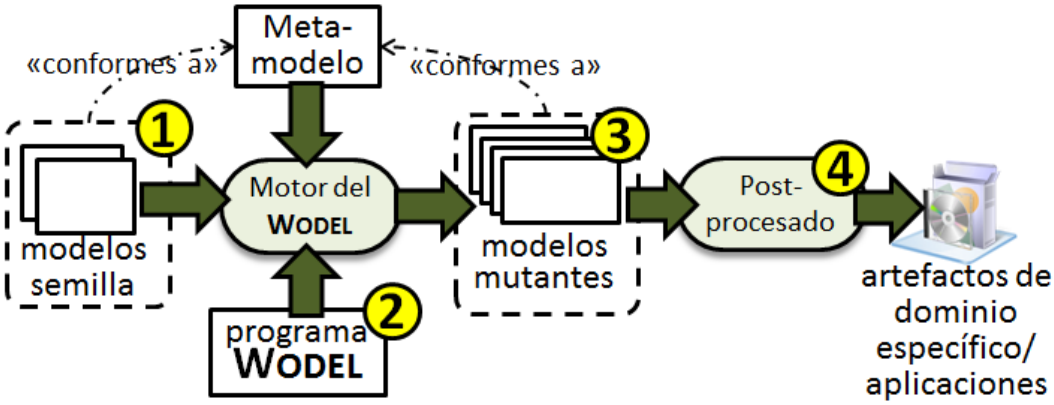

Figura 4.1 Esquema de nuestra propuesta

La ejecución de un programa Wodel genera mutantes de los modelos semilla (etiqueta 3). Estos modelos mutantes son también modelos válidos (son conformes al meta-modelo de los modelos semilla), ya que Wodel lo comprueba cada vez que se va a generar un mutante. Si el modelo mutante que se va a generar no es conforme al meta-modelo, Wodel repetirá el proceso un número máximo *n* de veces, configurable por el usuario, hasta conseguir un mutante válido. En caso de no lograr generar un mutante válido en ese número máximo *n* de intentos, no se genera nada, y se sigue la ejecución normal del programa. Finalmente, puede utilizarse un paso opcional de post-procesado para generar artefactos específicos de un dominio para aplicaciones particulares de los mutantes (etiqueta 4).

4.2. El DSL Wodel

En esta sección se presenta el DSL Wodel y se muestra su utilización mediante ejemplos de operadores de mutación sobre autómatas finitos conformes al meta-modelo de la Figura 2.1.

Un programa Wodel consta de dos partes. La primera parte consiste en la declaración opcional del número de mutantes que se generan, valor que si se omite se toma por defecto de las preferencias, la carpeta de salida, los modelos semilla y su meta-modelo. La segunda parte consiste en la definición de los operadores de mutación y cuántas veces se van a aplicar. De forma opcional, se puede indicar varias secciones de generación de mutantes, para controlar el momento en que estos mutantes son generados, y dentro de esta estructura de secciones, se puede también indicar si los modelos semilla de cada sección corresponden a los mutantes de salida de una o varias de las secciones anteriores.

El Listado 4.1 muestra un programa Wodel sencillo. La línea 1 establece que se quieren generar 2 mutantes en la carpeta *out*, a partir del modelo semilla *evenBinary.fa*. La línea 2 indica el meta-modelo del modelo semilla. Las líneas 4-8 definen tres operadores de mutación: el primer operador (línea 5) selecciona un estado final aleatorio, y lo actualiza a estado no final; el segundo (línea 6) crea un

nuevo estado final; y el último (línea 7) crea una nueva transición desde el estado seleccionado en la línea 5 al nuevo estado creado en la línea 6.

```
1. generate 2 mutants in "out/" from "evenBinary.fa"
2. metamodel "http://fa.com"
3.
4. with commands {
5.      s0 = modify one State where {isFinal = true} with {reverse(isFinal)}
6.      s1 = create State with {isFinal = true}
7.      t0 = create Transition with {src = s0, tar = s1, symbol = one Symbol}
8. }
```

Listado 4.1 Un programa Wodel sencillo

A continuación, se detallan las primitivas de mutación que ofrece Wodel. Estas primitivas incluyen operaciones atómicas para crear y eliminar objetos y referencias, modificar valores de atributos, o redirigir el origen o el destino de las referencias. La Figura 4.2 muestra un extracto del meta-modelo del Wodel con la definición de algunas primitivas de mutación representativas para el caso de estudio (autómatas finitos deterministas). Todas las clases de mutaciones heredan de *Mutation*, que contiene el número mínimo y máximo de veces que debe aplicarse la mutación. Si se omite la información, como en las mutaciones del Listado 4.1, las mutaciones se ejecutan una vez. Un objeto *Mutation* es a su vez un *ObjectEmitter* que puede tener un nombre, de forma que puede referenciarse desde otras instrucciones de mutación. Por ejemplo, en la línea 7 del Listado 4.1, se utiliza el nombre *s0* para referenciar el *State* modificado en la línea 5. Los tipos principales de *Mutation* son los siguientes:

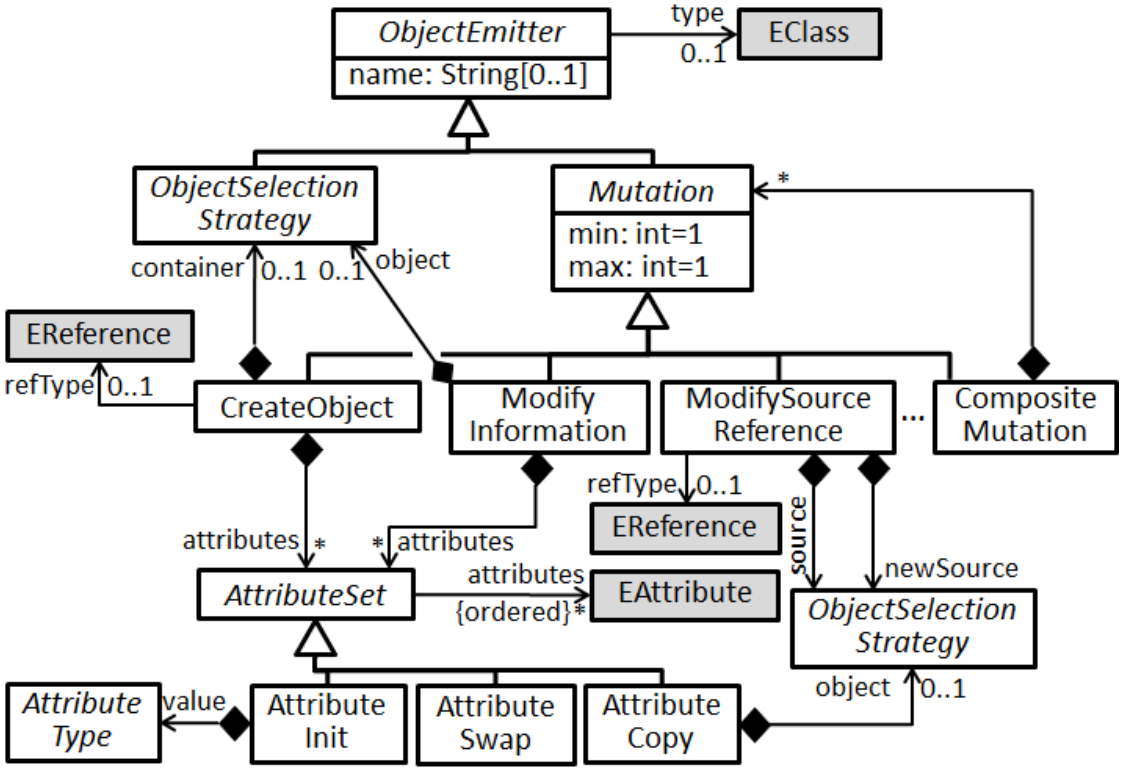

Figura 4.2 Fragmento del meta-modelo Wodel: primitivas de mutación

- *CreateObject*: crea el objeto de la clase indicada por la referencia *type*. De forma opcional, es posible seleccionar un objeto contenedor para el objeto creado utilizando una *ObjectSelectionStrategy* (explicada a continuación). En ese caso, *refType* indica la referencia al contenedor en el que se colocará el nuevo objeto. Si no se indica el objeto contenedor, Wodel selecciona uno del tipo adecuado, y si existen varios, elige uno de forma aleatoria. En la línea 6 del Listado 4.1, no es necesario especificar un contenedor para el nuevo *State* porque, asumiendo un *Automaton* por modelo, el estado creado sólo puede colocarse en la colección *states* de *Automaton*. De forma alternativa, se puede indicar de forma explícita el objeto contenedor utilizando `create State in one Automaton.states`. De forma similar, es posible especificar un valor para los atributos y referencias del objeto creado, y en caso de que no se indique ningún valor para una referencia obligatoria, Wodel le asigna un objeto de un tipo compatible. Las clases *EClass*, *Ereference* y *EAttribute* que aparecen sombreadas en la figura son las que proporciona Ecore para la definición de meta-modelos. En este ejemplo, *Automaton* es una *EClass*, *states* es una *EReference*, y *name* es un *EAttribute*. Esta forma de refererirse a los elementos del dominio del meta-modelo permite la comprobación de tipos y la asistencia de contenido.

- *CreateReference*: crea una nueva referencia del tipo dado entre dos objetos. Los objetos pueden seleccionarse utilizando una *ObjectSelectionStrategy*, en caso de no indicarse una, se eligen de forma aleatoria los objetos origen y destino del tipo adecuado.

- *ModifyInformation*: selecciona un objeto mediante una *ObjectSelectionStrategy*, y proporciona un conjunto de modificaciones para aplicarlas sobre sus atributos (clase *AttributeSet*). El meta-modelo muestra sólo algunas de las modificaciones posibles, como la inicialización del valor de un atributo, intercambio del valor de dos atributos o referencias, o copia del valor de un atributo en otro. Otras modificaciones dependen del tipo del atributo. Por ejemplo, se puede invertir el valor de atributos booleanos (como se hace en la línea 5 del Listado 4.1), y los atributos de tipo string pueden ponerse en mayúsculas o en minúsculas, ser sustituidos por una cadena elegida de forma aleatoria de un conjunto dado, o sustituir alguna parte del string por otra.

- *ModifySourceReference*, *ModifyTargetReference*: redirige el origen o el destino de una referencia por otro objeto seleccionado mediante una *ObjectSelectionStrategy*.

- *RemoveObject*: Elimina de forma segura un objeto seleccionado por una *ObjectSelectionStrategy*, asegurando que no queda suelta ninguna referencia origen o destino al objeto eliminado.

- *RemoveReference*: Elimina una referencia de un tipo dado. Los objetos de origen y de destino de la referencia se pueden definir utilizando una *ObjectSelectionStrategy*.

- *CompositeMutation*: Permite definir mutaciones compuestas que constan de una secuencia de mutaciones atómicas o de otras mutaciones compuestas, que se ejecutan en bloque.

De forma adicional, la operación *Select* permite seleccionar objetos o referencias de acuerdo a un criterio, de forma que pueden utilizarse en mutaciones posteriores.

La selección de objetos y referencias en las mutaciones y los selectores puede hacerse utilizando las siguientes estrategias: seleccionar un elemento aleatorio, un elemento específico (referenciado por el nombre de un emisor), todos los elementos que satisfacen cierta condición, o un elemento diferente del seleccionado en la mutación activa. El meta-modelo en la Figura 4.3 muestra tres de estas estrategias: *SpecificObjectSelection* selecciona un objeto referenciado por un emisor; *SpecificReferenceSelection* selecciona a la vez un objeto y una referencia definidos por la clase del objeto; y *RandomObjectSelection* elige un objeto aleatorio de la clase especificada por la referencia *type*. Todas las estrategias pueden parametrizarse con una *condition* (clase *Expression*) sobre los valores de atributos y referencias del elemento seleccionado. Por ejemplo, en la línea 5 del Listado 4.1, la mutación *ModifyInformation* utiliza una estrategia *RandomObjectSelection* (`one State`) con una condición sobre atributos (`where {isFinal = true}`).

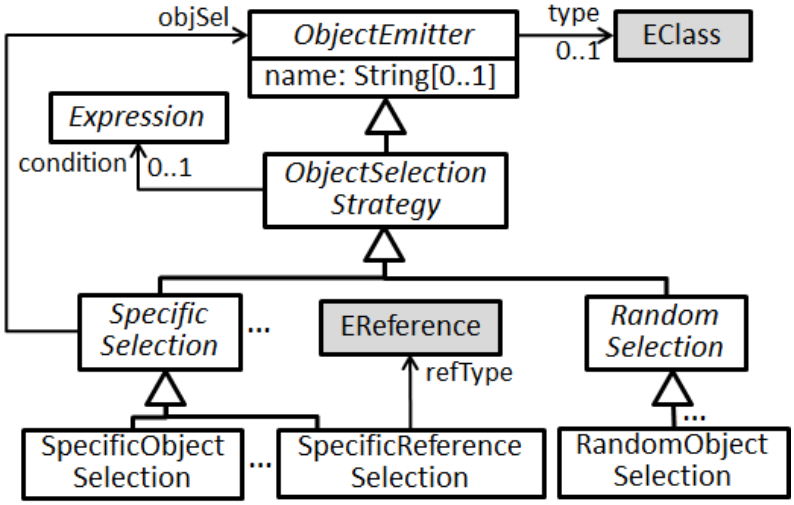

Figura 4.3 Fragmento del meta-modelo Wodel: estrategias de selección

Wodel permite tomar los mutantes generados en una instancia previa como modelos semilla de mutaciones posteriores, como se muestra en el Listado 4.2. Wodel permite realizar estos bloques de mutación mediante la palabra reservada *blocks* (línea 4). Se puede indicar el número máximo de mutantes que van a generarse en cada bloque de forma exclusiva (líneas 7 y 10). Un bloque de mutaciones definida así puede tomar como modelos semilla los mutantes generados en bloques previos mediante la palabra clave *from* (línea 8), o los especificados en el comienzo del fichero si no se indica ningún bloque base. Wodel permite además indicar que no se repitan los mutantes que se generan en estos bloques asegurando su unicidad dentro de ese bloque, y con respecto a los bloques a partir de los que se han tomado los modelos semilla, con la directiva *repeat=no* (línea 8). Las instrucciones de mutación incluidas en estos bloques cambian el destino de una transición elegida de forma aleatoria

(línea 6), e invierten el valor del atributo booleano *isFinal* de un estado elegido de forma aleatoria (línea 9).

También se pueden incluir restricciones OCL dentro del código Wodel mediante la palabra clave *constraints*, indicando la clase sobre la que se debe aplicar la restricción. Estas restricciones adicionales se aplican sobre los mutantes, y no se han especificado necesariamente en su meta-modelo. En este ejemplo se muestra una restricción OCL que asegura que los elementos *State* del autómata mutado están todos conectados entre sí (líneas 12-16).

```
1. generate mutants in "out/" from "model/"
2. metamodel "http://fa.com"
3.
4. with blocks {
5.       first {
6.               modify target tar from one Transition to other State
7.       } [2]
8.       second from first repeat=no{
9.               modify one State with {reverse(isFinal)}
10.      } [3]
11. }
12. constraints {
13.      context State connected: "isInitial or Set{self}->closure
14.              (s | Transition.allInstances()->select(t | t.tar=s)
15.              ->collect(src))->exists(s | s.isInitial)"
16. }
```

Listado 4.2 Un programa Wodel con bloques y una restricción OCL

Como se ha visto, Wodel tiene una sintaxis concreta textual. El Listado 4.3 muestra un extracto de su gramática.

```
1.  WODELPROGRAM ::= DEFINITION with ( blocks { BLOCK* } | commands { MUTATION* } )
2.  ( constraints { CONSTRAINT* } );
3.
4.  DEFINITION ::=
5.  generate (<num>)? mutants in <folder> from SEEDS
6.  metamodel <meta-model>
7.
8.  BLOCK ::= <name> ( from <name-block> (, <name-block>)* )?
9.  (repeat = REPEAT)? { MUTATION* } ( [ (<min> ..)? <max> ] )?
10.
11. MUTATION ::=
12. ( CREATEOBJECT | MODIFYINFORMATION |
13. MODIFYSOURCEREFERENCE | ... | COMPOSITEMUTATION )
14. ( [ (<min> ..)? <max> ] )?
15.
16. CREATEOBJECT ::=
17. ( <name> '=' )? create <EClass>
18. ( in OBJECTSELECTIONSTRATEGY ( '.' <EReference> )? )?
19. ( with { ATTRIBUTESET ( , ATTRIBUTESET )* } )?
20.
21. MODIFYINFORMATION ::=
22. ( <name> '=' )? modify OBJECTSELECTIONSTRATEGY
23. with { ATTRIBUTESET ( , ATTRIBUTESET )* }
24.
25. MODIFYSOURCEREFERENCE ::=
26. modify source <EReference>
27. ( from OBJECTSELECTIONSTRATEGY )?
28. ( to OBJECTSELECTIONSTRATEGY )?
29.
30. COMPOSITEMUTATION ::= ( <name> '=' )? [ MUTATION* ]
31. ...
```

Listado 4.3 Extracto de la gramática de Wodel

El Listado 4.4 muestra otro ejemplo que genera 3 mutantes de todos los modelos en la carpeta *model* (línea 1). Las líneas 5-7 definen una mutación compuesta que elimina un estado no inicial (línea 5), y a continuación elimina todos los objetos *Transition* que le apuntan, o que salen del estado eliminado (líneas 6-7). Las transiciones a eliminar son aquellas que tienen un valor indefinido en las referencias *src* o *tar*. La mutación en la línea 8 selecciona un objeto *Transition* de forma aleatoria, y modifica su referencia *symbol* para apuntar a otro objeto *Symbol* diferente. Las mutaciones definen una cardinalidad, de forma que la primera se aplicará de 0 a 2 veces en cada mutante, y la segunda, de 1 a 3 veces.

```
1. generate 3 mutants in "out/" from "model/"
2. metamodel "http://fa.com"
3.
4. with commands {
5.      c0 = [ remove one State where {isInitial = false}
6.             remove all Transition where {src = null}
7.             remove all Transition where {tar = null} ] [0..2]
8.      modify target symbol from one Transition to other Symbol [1..3]
9. }
```

Listado 4.4 Mutaciones compuestas y cardinalidades

Como los programas Wodel manejan cada mutación definida como una operación, no es posible que aparezcan dos mutaciones contradictorias; de todas formas, dos mutaciones pueden anularse entre sí (p. ej., una mutación crea un objeto, y otra lo elimina).

4.2.1. Expresividad y concisión de Wodel

A continuación, se analiza la expresividad y concisión de Wodel. La expresividad se evalúa utilizando Wodel para definir mutaciones interesantes para autómatas, bien diseñadas por nosotros, o bien encontradas en la literatura [25]. Hay que señalar que en [25] no se consideran estados finales ni se mutan símbolos de transiciones, pero se puede aplicar un número variable de mutaciones que en Wodel se expresa con las cardinalidades.

La Tabla 4.1 lista las mutaciones. Las primeras doce cambian el lenguaje que reconoce un autómata (asumiendo que es mínimo), y las dos últimas convierten al autómata en indeterminista. Aunque Wodel no tiene todas las típicas estructuras de control de los lenguajes de programación de propósito general, su expresividad es suficiente para indicar las mutaciones del ejemplo que se presenta. Además, se pueden emular bucles mediante mutaciones compuestas y cardinalidades, y los condicionales están implícitos en las estrategias de selección. Aunque la expresividad de Wodel no depende del meta-modelo para el que se definen las mutaciones, su utilización en otros contextos de aplicaciones (p. ej., pruebas de transformación de modelos) podría requerir incluir nuevas primitivas de Wodel. La aplicación de Wodel a otros dominios es trabajo futuro.

Mutaciones que cambian el lenguaje

Crear transición [25]	`create Transition with {symbol = one Symbol}`
Crear estado final	Listado 4.1, línea 6

Crear estado conectado	`s = create State`
	`  with {name = random-string(1,4)}`
	`t = create Transition`
	`  with {tar = s, symbol = one Symbol}`

Eliminar transición	`remove one Transition`
Eliminar estado y transiciones adyacentes	Listado 4.4, líneas 5-7
Cambiar símbolo en transición	Listado 4.4, línea 8
Cambiar estado final a no final	Listado 4.1, línea 5

Cambiar el estado inicial a otro diferente [25]	`s0 = modify one State where {isInitial = true}`
	`  with {isInitial = false}`
	`s1 = modify one State where {self <> s0}`
	`  with {isInitial = true}`

Invertir la dirección de una transición [25]	`modify one Transition with {swap(src, tar)}`

Intercambiar el símbolo de dos transiciones con el mismo estado origen	`t = select one Transition`
	`modify one Transition`
	`  where {self <> t and src = t.src}`
	`  with {swap(symbol, t.symbol)}`

Redirigir una transición a un nuevo estado final	`s = create State`
	`  with {name = 'f', isFinal = true}`
	`modify target tar from one Transition to s`

Combinación de añadir una nueva transición y cambiar el estado inicial [25].	`s0 = modify one State where {isInitial = true}`
	`  with {reverse(isInitial)}`
	`s1 = modify one State where {self <> s0}`
	`  with {isInitial = true}`
	`create Transition`
	`  with {src = s1, tar=s0,`
	`  symbol = one Symbol}`

Mutaciones que producen un autómata indeterminista

Crear una transición λ	`create Transition`

Crear una transición con el mismo símbolo desde un estado a otro diferente	`t = select one Transition where {symbol <> null}`
	`create Transition`
	`  with { src = t.src, symbol = t.symbol,`
	`  tar = one State where {self <> t.tar}}`

Tabla 4.1 Utilización de Wodel para definir mutaciones de autómatas

Para evaluar la concisión de Wodel, se ha comparado el código de un programa Wodel con el código equivalente en Java, que sería la alternativa inmediata para integrar operaciones de mutación en aplicaciones. Programar los operadores de mutación en Java requiere conocimientos de la API reflexiva de EMF [29], ya que no se puede contar con que las clases de implementación en Java existen para los tipos en el meta-modelo dado. También se requiere tener cuidado con los detalles

accidentales que Wodel maneja de manera transparente, como colocar objetos en contenedores, inicializar referencias obligatorias, comprobar tipos con relación al meta-modelo, serializar modelos, comprobar que los mutantes generados están bien formados, o comparar mutantes generados para evitar que haya duplicados.

Para mostrar la complejidad del código Java equivalente, el Listado 4.5 muestra parte del código necesario para implementar la mutación `create Transition`. Este fragmento crea una transición (líneas 3-4), obtiene un objeto *Automaton* de los modelos semilla (líneas 8-10), añade la transición al autómata (líneas 12-13), selecciona un estado aleatorio (líneas 15-17), y establece este estado como origen de la transición creada (líneas 18-19). El listado omite el código para las tareas como carga del modelo, o la comprobación de que el resultado es válido. En total suma 103 líneas de código, sin contar saltos de línea ni comentarios. En su lugar, la misma instrucción sólo requiere 1 línea utilizando Wodel.

```
1. ...
2. // create transition
3. EClass transitionClass = (EClass)epackage.getEClassifier("Transition");
4. EObject transition = EcoreUtil.create(transitionClass);
5.
6. // search object automaton in model
7. EObject automaton = null;
8. for (TreeIterator<EObject> it = seed.getAllContents(); it.hasNext();) {
9.     automaton = it.next();
10.    if (automaton.eClass().getName().equals("Automaton")) {
11.        // add transition to automaton
12.        EStructuralFeature feature = automaton.eClass().
               getEStructuralFeature("transitions");
13.        ((List<EObject>)automaton.eGet(feature)).add(transition);
14.        // set random state as source of the transition
15.        feature = automaton.eClass().getEStructuralFeature("states");
16.        List<EObject> states = (List<EObject>)automaton.eGet(feature);
17.        EObject randomState = states.get(rand.nextInt(states.size()));
18.        feature = transitionClass.getEStructuralFeature("src");
19.        transition.eSet(feature, randomState);
20. ...
```

Listado 4.5 Código Java para la mutación *create Transition*

Wodel se puede comparar también con otros lenguajes de transformación de modelos, como por ejemplo, con lenguajes de reglas de transformación de grafos [40].

4.3. Registro de mutaciones

Se realiza el diseño e implementación de una extensión al entorno del Wodel con el objetivo de registrar las mutaciones aplicadas sobre los modelos semilla, de forma que se pueda utilizar en las extensiones de post-procesado, para p. ej., presentar al usuario opciones de texto con las instrucciones de mutación aplicadas, como se muestra en la generación de ejercicios complejos de Wodel-Edu en la Sección 5.

Para manejar este registro de mutaciones aplicadas se utilizan también modelos de EMF, de forma que para cada mutante generado, se genera a su vez un modelo con el registro de mutaciones aplicadas en ese mutante. La Figura 4.4 muestra un extracto del meta-modelo de este registro de mutaciones.

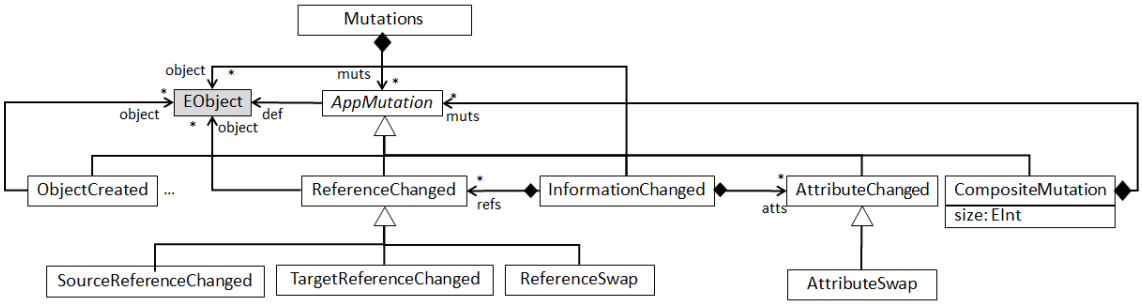

Figura 4.4 Extracto del meta-modelo del registro de mutaciones aplicadas

El objeto raíz queda definido por la clase *Mutations*, que contiene elementos de tipo *AppMutation*. Estos elementos *AppMutation* guardan la información de las instrucciones de mutación aplicadas por Wodel: la referencia *def* apunta a la instrucción de mutación correspondiente del fichero de código del programa Wodel. Esto se realiza mediante la serialización del código que proporciona Xtext, de forma que los ficheros de código de los DSLs implementados con este plug-in se convierten a modelos XMI de EMF, permitiendo trabajar con ellos programáticamente. Esta generación del registro de mutaciones es opcional, y puede activarse o desactivarse desde la pantalla de preferencias del IDE del Wodel. Según sea el tipo de la instrucción de mutación, se crea el elemento correspondiente, p. ej., para una instrucción *CreateObject* en el programa Wodel, se crea el elemento *ObjectCreated* en el modelo del registro, que guarda la referencia *def* a la instrucción de mutación del programa Wodel, y la referencia *object* al objeto creado. Dentro de la generación de este registro de mutaciones hay referencias a los elementos del modelo semilla (p. ej., los elementos eliminados del modelo semilla que ya no están en el modelo mutante), y otros que apuntan al modelo mutante que se obtiene una vez realizadas las operaciones de mutación (p. ej., los elementos creados en el mutante que no estaban en el modelo semilla). Se implementa además una extensión de post-procesado para compactar estos modelos de registro, de forma que si, p. ej., hay instrucciones de mutación que se anulan entre sí (p. ej., se crea y se elimina el mismo elemento), estas no aparecen en el registro, ya que son irrelevantes. Esta compactación del registro de mutaciones aplicadas también es opcional, y se puede configurar desde la pantalla de preferencias del IDE de Wodel.

4.4. Herramienta proporcionada

Se ha implementado un entorno de desarrollo para Wodel, disponible como plug-in de Eclipse [8], para mutar modelos EMF. La Figura 4.5 muestra su arquitectura. El entorno proporciona un editor para Wodel, hecho con Xtext, que incorpora un validador, y otras facilidades de completado de código para ayudar a los usuarios a seleccionar clases, referencias y atributos válidos del dominio del meta-modelo. Los programas Wodel correctos se compilan automáticamente a código Java mediante un generador de código hecho en Xtend. El código Java generado, que se encarga de generar los mutantes a partir de los modelos semilla, puede ejecutarse de forma transparente desde el IDE de Wodel. La ventaja de generar código Java de forma explícita es que puede utilizarse en aplicaciones autónomas. Además, este código es genérico ya que manipula los modelos utilizando la reflexividad y, por tanto, puede utilizarse para mutar cualquier modelo conforme al meta-modelo del domino (en el Listado 4.5 se muestra un ejemplo de la API reflexiva de EMF). Como añadido, Wodel define un punto de extensión que permite a los usuarios registrar post-procesadores específicos del dominio para ejecutarse después de la generación de los mutantes (Wodel-Edu aprovecha esta funcionalidad, como se muestra en la Sección 5).

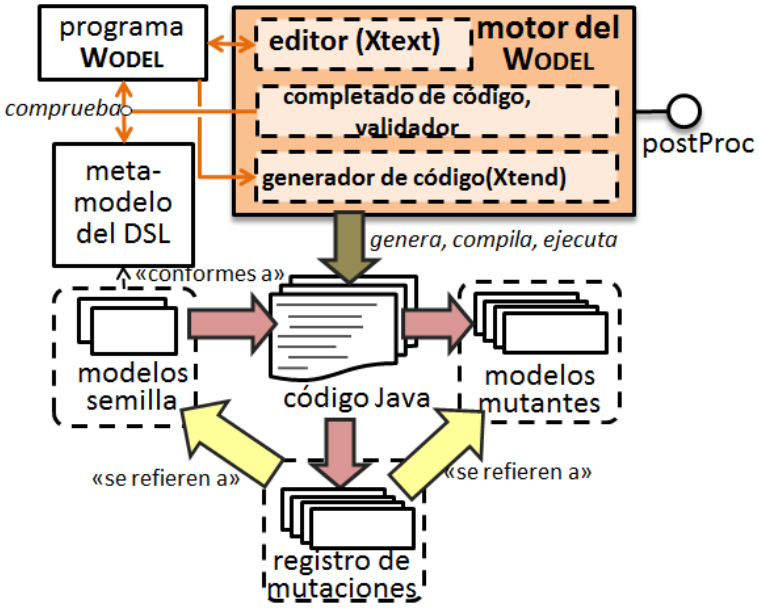

Figura 4.5 Arquitectura del entorno del Wodel

La Figura 4.6 muestra una captura de pantalla del IDE, con alguna de las facilidades de completado de código. En este caso, el editor sugiere atributos válidos para la clase *State*, y algunos operadores de modificación que se pueden aplicar.

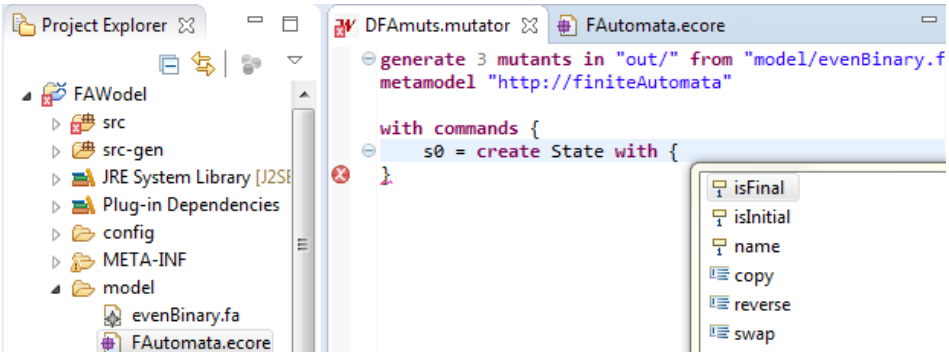

Figura 4.6 Captura de pantalla del IDE del Wodel

En la Figura 4.7 se muestra la estructura de carpetas con los modelos mutantes de salida para el programa Wodel del Listado 4.2. Se puede observar que Wodel crea una carpeta para cada modelo semilla en la que almacenar los mutantes que corresponden a ese modelo semilla, para ello crea una carpeta para cada bloque definido en el código, en caso de que los hubiera, y en el caso de que haya bloques que toman como modelos semilla los mutantes de salida de otros bloques, los clasifica por carpetas según el nombre del bloque de origen. También se muestra que genera el modelo de registro correspondiente para cada modelo mutante de salida.

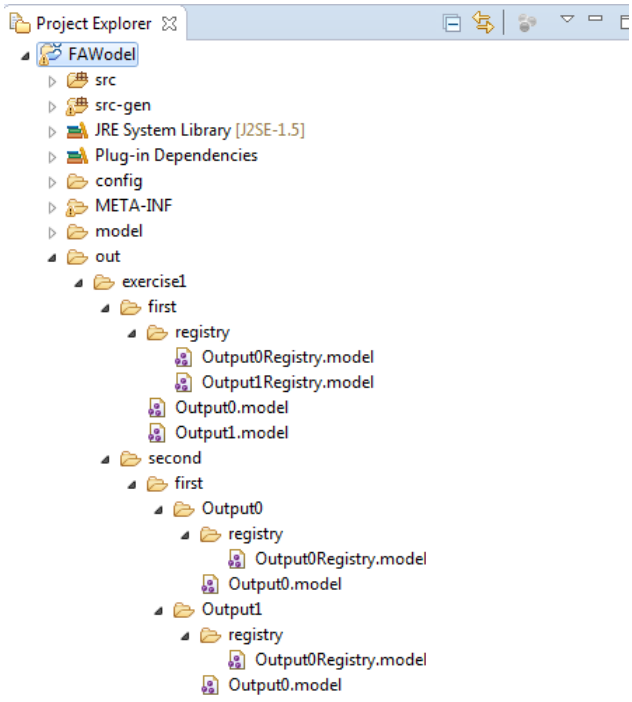

Figura 4.7 Estructura de carpetas de los mutantes de salida para el programa Wodel del Listado 4.2

Capítulo 5. Generación automática de ejercicios con Wodel-Edu

Wodel-Edu es una extensión de post-procesado de Wodel para la generación automática de ejercicios. Los ejercicios se corrigen de forma automática y pueden servir a los estudiantes para auto-evaluación. La utilización de técnicas de mutación para generación de ejercicios que se corrigen de forma automática consiste en la idea siguiente: dado un modelo que es solución a un ejercicio de un determinado dominio, se pueden aplicar sucesivas mutaciones sobre este modelo. En cada paso en el que se aplican las mutaciones, éstas quedan registradas, teniendo el sistema toda la información necesaria: las mutaciones que se han aplicado en cada paso, y el modelo semilla sobre el que se han aplicado estas mutaciones. Contando con esta información, el sistema es capaz de identificar en cada modelo mutante cuáles son las instrucciones de mutación que se han aplicado, y de esta forma identificar cuáles corresponden al modelo mutante que se presenta en el ejercicio.

Con esta información, Wodel-Edu genera ejercicios de tres tipos diferentes. El primer y el segundo formato son ejercicios sencillos. El primer formato consiste en que la aplicación muestra al estudiante varios diagramas entre los que sólo uno es correcto; el estudiante ha de seleccionar cuál es. En este tipo de ejercicios basta con presentar al estudiante un modelo semilla (sin mutaciones) entre otros modelos mutados. En el segundo formato de ejercicios, se muestra al estudiante un diagrama. El estudiante ha de indicar si es correcto o no. En este caso se presenta al estudiante un modelo elegido aleatoriamente, en caso de que sea un modelo semilla, el modelo es correcto, y en caso de que sea un modelo mutante, el modelo es incorrecto. El tercer formato de ejercicios es más complejo, consiste en presentar al estudiante un diagrama, que se ha generado aplicando una o varias mutaciones sobre un modelo semilla. Se le muestran además varias opciones de texto a elegir, correctas e incorrectas, y el estudiante ha de descubrir cuáles son las correctas. En este tercer formato de ejercicios, las opciones correctas se generan a partir de las instrucciones de mutación aplicadas para generar el modelo mutante que se muestra en pantalla, y las opciones incorrectas se generan a partir de las instrucciones de mutación que se

aplican para crear otros modelos mutantes que toman como semilla el modelo mutante presentado al estudiante.

Para ello, Wodel-Edu requiere de lo siguiente: configuración de los ejercicios a presentar en la aplicación web y estructura del test; visualización gráfica de los modelos; configuración de las opciones para el tipo de ejercicios complejos; e identificación de los elementos del modelo. Todas estas características se implementan mediante DSLs, aunque se generan opciones por defecto. En primer lugar, la Sección 5.1 presenta la arquitectura de Wodel-Edu, implementado como extensión al entorno del Wodel. En segundo lugar, la Sección 5.2 presenta los DSLs utilizados por Wodel-Edu para la generación automática de ejercicios. En tercer lugar, la Sección 5.3 presenta un ejemplo de ejercicio generado, y el procedimiento para generar los ejercicios. La Sección 5.4 describe la herramienta proporcionada. Por último, la Sección 5.5 presenta los resultados de la evaluación realizada sobre una aplicación de ejercicios generada con Wodel-Edu.

5.1. Arquitectura

La Figura 5.1 muestra la arquitectura de Wodel-Edu. Para permitir la configuración de ejercicios para distintos dominios (p. ej., autómatas finitos, diagramas de clases, etc.), el post-procesado puede configurarse mediante cuatro DSLs: el DSL *eduTest* para la descripción del estilo de los ejercicios; el DSL *modelDraw* para la representación gráfica (o sintaxis concreta) de los elementos; el DSL *idModel* para definir cadenas de texto con las que identificar los elementos del modelo; y el DSL *configOptions* para configurar las opciones de respuesta que se presentan en los ejercicios de test de la aplicación. Los DSLs *idModel* y *configOptions* se utilizan en el tercer formato de ejercicios. En este capítulo se ilustrará el uso de estos cuatro lenguajes para la configuración de Wodel-Edu para la generación de ejercicios de autómatas finitos, pero Wodel-Edu es también independiente del meta-modelo, y puede utilizarse con otros dominios. Los modelos de descripción de los ejercicios y representación de los elementos se utilizan siempre, siendo opcional la utilización de la identificación de los elementos, y la configuración de opciones.

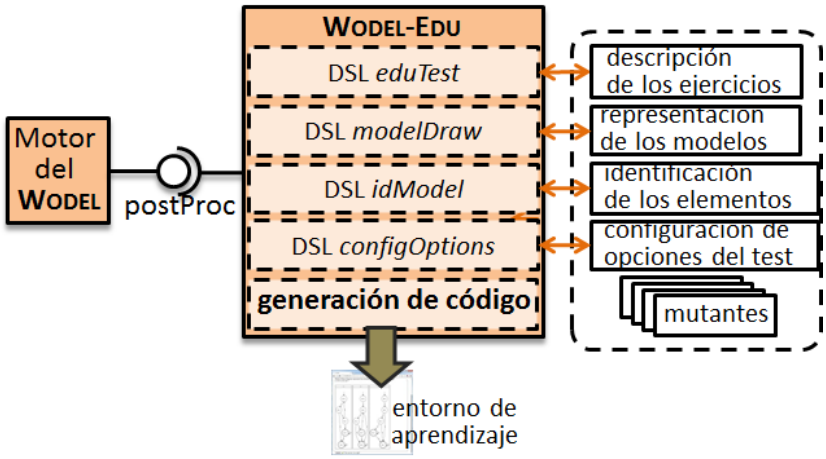

Figura 5.1 Arquitectura del plug-in Wodel-Edu

5.2. Lenguajes de configuración de ejercicios

La Figura 5.2 muestra el meta-modelo del DSL *eduTest*. Este DSL genera el código html de las pantallas de ejercicios para cada bloque que se haya definido en el DSL Wodel (referencia *block* de la clase *MutatorTests*). *Program* es la clase raíz, tiene una configuración *config* para deshabilitar la navegación entre ejercicios de tal forma que una vez contestado un ejercicio no es posible volver a mostrarlo (*navigation=locked*), o habilitarla (*navigation=free*). La clase *Program* contiene elementos de la clase *MutatorTests*, que pueden utilizar los mutantes generados en cada bloque definido en el programa *Wodel* mediante la referencia *block*. Los elementos *MutatorTests* pueden ser de dos tipos: *SimpleSelection*, para los ejercicios sencillos; y *SelectCorrection*, para los ejercicios complejos. Cada una de estas subclases de *MutatorTests* tiene diferentes parámetros de configuración, definidos en las clases *SimpleConfiguration* y *SelectConfiguration*, respectivamente.

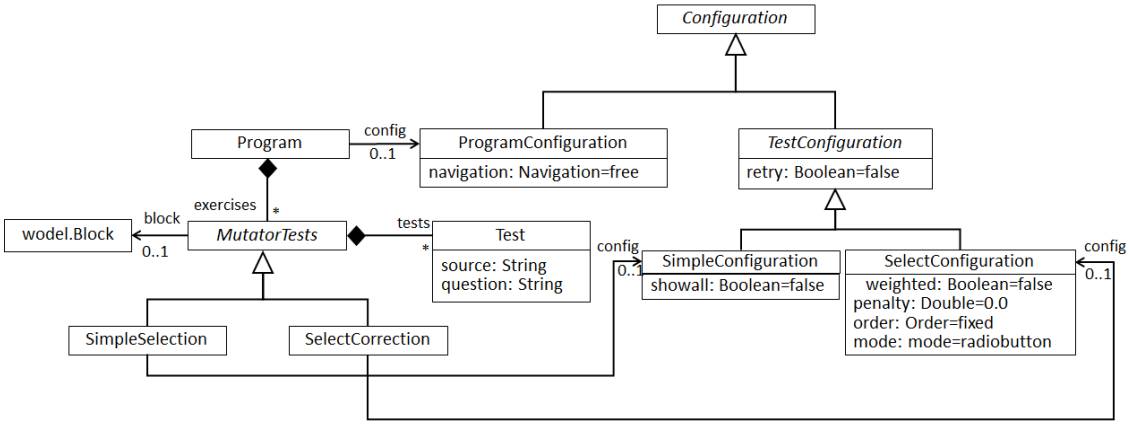

Figura 5.2 Meta-modelo del DSL *eduTest*: descripción de los ejercicios

El Listado 5.1 muestra una parte de la descripción de algunos ejercicios, utilizando el DSL *eduTest*. La línea 2 indica que se puede navegar libremente por las páginas de la aplicación, retrocediendo a páginas de ejercicios que ya se hayan completado, y avanzando para consultar o empezar por otras páginas de ejercicios. La línea 3 indica que se va a utilizar el bloque *first* para la generación de ejercicios complejos. Este tipo de ejercicios tiene varios parámetros de configuración: *retry*, para habilitar los reintentos en caso de fallo; *weighted*, para indicar si los ejercicios tendrán un peso diferente según su dificultad (este parámetro de la dificultad se calcula según el número de opciones que se presentan en el ejercicio, que no tienen por qué ser el mismo para todos, p. ej., si se repite la misma instrucción de mutación, Wodel-Edu sólo muestra una opción para esa mutación); *penalty*, para incluir un ratio de penalización por ejercicio fallado; *order*, para ordenar los ejercicios según el número de opciones que se muestran, de manera descendente o ascendente, según su aparición en el código del programa *eduTest*, o con un orden aleatorio; y *mode* que indica si las opciones se agruparán según los bloques, de forma que sólo una de ellas es correcta (*radiobutton*), o si se presentarán las distintas opciones desglosadas, de forma que pueda haber más de una respuesta correcta (*checkbox*).

```
1. Tests {
2.     navigation=free
3.     SelectCorrection first {
4.             retry=yes, weighted=no, penalty=0.0,
5.             order=mutations descending, mode=radiobutton
6.             description for 'exercise4.model' = 'The automaton for language a+b+...'
7.             description for 'exercise6.model' = 'The automaton for language a*b...'
8.             ...
9.     }
10.    SimpleSelection simple {
11.            retry=no, showall=yes
12.            description for 'exercise1.model' = 'Select which of these automata ...'
13.            description for 'exercise2.model' = 'Select which of these automata ...'
14.            ...
15.    }
16. }
```

Listado 5.1 Sintaxis concreta textual del DSL *eduTest*

La línea 10 indica que se van a generar ejercicios sencillos a partir del bloque *simple*. Este tipo de ejercicios también permite habilitar reintentos o bloquearlos, y se pueden generar ejercicios de dos tipos mediante la directiva *showall* (línea 11): el primero (`showall=yes`), muestra varios diagramas al usuario y éste debe seleccionar cuál es el correcto; el segundo (`showall=no`), muestra un único diagrama al usuario, y éste debe decidir si el diagrama es correcto, o no.

En la Figura 5.3 se presenta el meta-modelo del DSL *modelDraw*, que permite definir la representación gráfica (sintaxis concreta gráfica) de los elementos de los modelos. Este DSL es independiente del meta-modelo del dominio. Esto se consigue mediante la utilización de las clases *EClass*, *EReference* y *EAttribute* de EMF. El objeto raíz es, en este caso, un objeto de tipo *MutatorGraph*, que tiene dos atributos: *metamodel* para indicar la ruta del meta-modelo del dominio; y *type* que indica el tipo de gráfico que se va a generar, por defecto de tipo *diagram*. De momento sólo se ha implementado este tipo de representación, queda como trabajo futuro aplicar este DSL para otros formatos. Un elemento de tipo *MutatorGraph* contiene además nodos, representados por la clase *Node*, y relaciones entre nodos, representados por la clase *Edge*. *MutatorGraph*, *Node* y *Edge*, heredan de la clase abstracta *Item*, que indica mediante la referencia *name* la *EClass* que corresponde a cada uno de los elementos. En el ejemplo de autómatas finitos, el atributo *name* de *MutatorGraph* es *Automaton*, el de *Node* es *State*, y el de *Edge* es *Transition*. La referencia *attribute* del elemento *Node* se utiliza para indicar un *EAttribute* booleano del elemento de tipo *EClass*, que debe tener valor *true* si el atributo *negation* del elemento *Node* está puesto a *false*. En resumen, esta referencia *attribute* sirve por tanto para representar de forma diferente *Nodes* que corresponden a los mismos elementos *EClass* pero tienen valores diferentes en sus atributos booleanos.

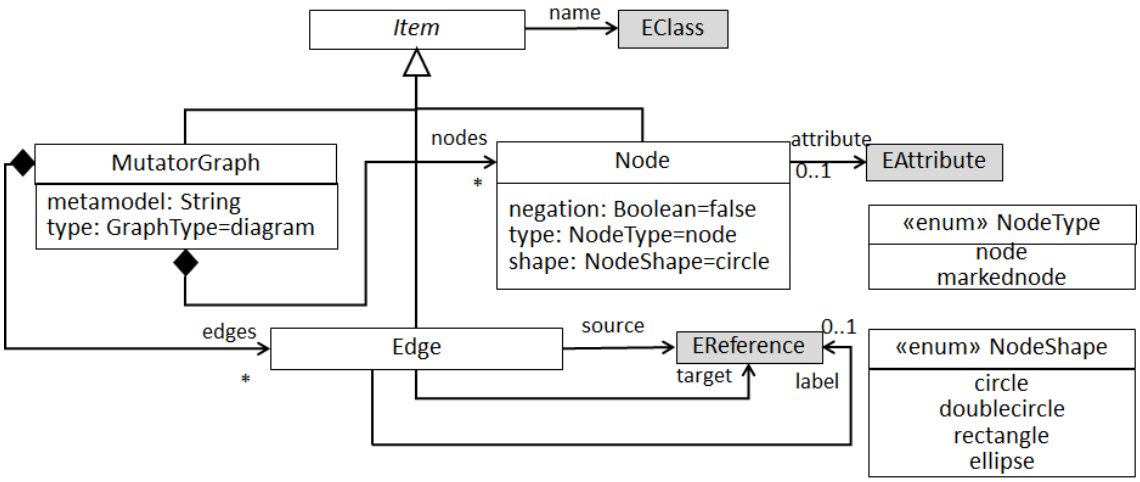

Figura 5.3 Meta-modelo del DSL *modelDraw*: representación gráfica de los modelos

El Listado 5.2 muestra un ejemplo del DSL *modelDraw* para la representación gráfica de los modelos. Este lenguaje es similar a la notación *dot* proporcionada por *Graphviz*[10], que es la tecnología que se emplea en Wodel-Edu para la visualización de los modelos. Este DSL proporciona al usuario un asistente de completado de código, indicando los valores que pueden utilizarse. Para ello, se indica dentro del código el meta-modelo del dominio (línea 1). *State(isInitial): markednode* (línea 4) indica que los elementos de tipo State con valor true en el atributo isInitial se representarán como un círculo marcado con una flecha. En la Figura 5.4 se muestra el diagrama generado que corresponde al modelo representado en la Figura 2.3.

```
1. metamodel "http://fa.com"
2.
3. Automaton: diagram {
4.     State(isInitial): markednode
5.     State(not isFinal): node, shape=circle
6.     State(isFinal): node, shape=doublecircle
7.     Transition(src, tar): edge, label=symbol
8. }
```

Listado 5.2 Sintaxis concreta textual del DSL *modelDraw*

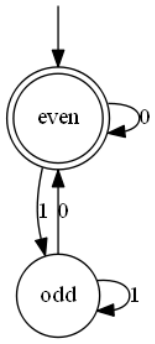

Figura 5.4 Diagrama generado por el DSL *modelDraw* para el modelo de la Figura 2.3

[10] http://www.graphviz.org/

En la Figura 5.5 se muestra el meta-modelo del DSL *idModel*. Este DSL se utiliza para la identificación de los elementos del modelo mediante la asignación de un texto representativo, y también es independiente del meta-modelo del dominio. La clase *IdentifyElements* define el objeto raíz del modelo, y puede contener objetos de tipo *Element*. Los objetos *Element* contienen a su vez elementos de tipo *Word*, que definen las palabras constantes *Constant*, y variables *Variable*, y tienen una referencia *type* de tipo *EClass*, que indica el elemento del meta-modelo del dominio que se quiere identificar. Pueden también tener una referencia *ref* a un objeto *EReference*, si se quiere especificar una cadena de texto para la referencia del objeto de tipo *type*. Además, también se puede incluir un atributo booleano, diferenciando de esta forma los elementos del modelo del dominio según los valores de los atributos especificados.

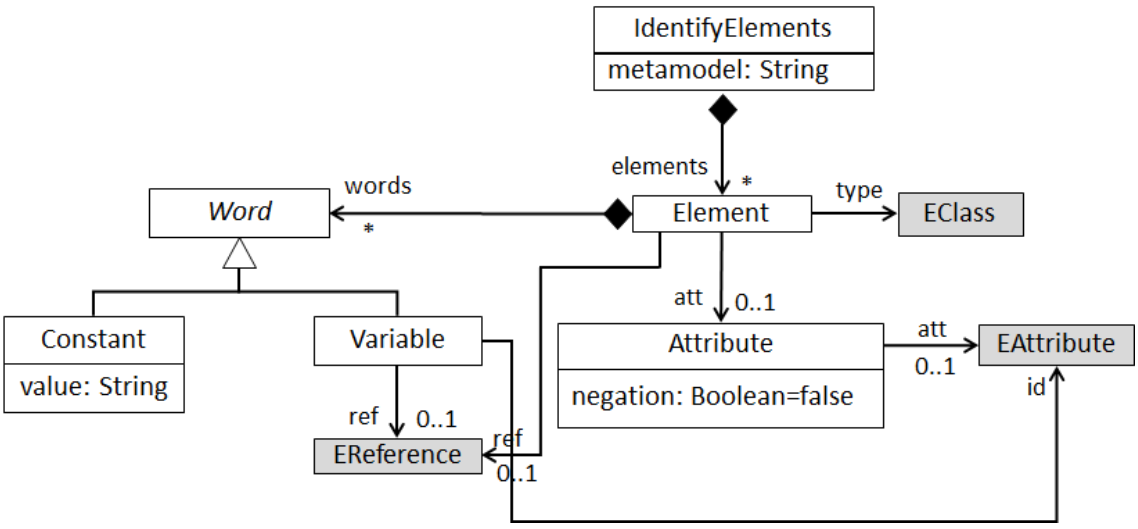

Figura 5.5 Meta-modelo del DSL *idModel*: identificación de los elementos del modelo

Se muestra un ejemplo de programa en el DSL *idModel* en el Listado 5.3. Este lenguaje se utiliza en Wodel-Edu para indicar al generador del código html de la aplicación web qué valores han de utilizarse para identificar a los elementos del modelo (p. ej., para identificar un objeto *State*, se puede indicar que se utilice su atributo *name*, ver línea 3). Además, este lenguaje proporciona la posibilidad de incluir otras cadenas de texto constantes que acompañen al valor o valores utilizados como identificadores, y también indicar el texto que se debe incluir para identificar referencias de los objetos (líneas 5 y 6). De la misma forma que en el DSL *modelDraw*, en *idModel* también se indica dentro del código el meta-modelo del dominio (línea 1), con el objetivo de presentar al usuario el asistente de completado de código.

```
1. metamodel "http://fa.com"
2.
3. >State: State %name
4. >Transition: Transition %symbol.symbol
5. >Transition.tar: target
6. >Transition.src: source
```

Listado 5.3 Sintaxis concreta textual del DSL *idModel*

Por último, la Figura 5.6 muestra el meta-modelo del DSL *configOptions*, mediante el que se puede configurar el texto que aparece en las opciones de los ejercicios complejos. La clase *Configuration* es la clase raíz y contiene elementos de tipo *Option*, que tienen una referencia *type* a una *EClass* (instrucción de mutación para la que se configura el texto de la opción que se va a presentar). Además, se puede también clasificar según el objeto sobre el que se aplica la mutación, especificando la referencia *object* de tipo *EObject* dentro del elemento *Option*. Un elemento *Option* tiene además dos referencias a elementos de tipo *Text*: *valid*, que indica el texto para la opción si es una solución al ejercicio; e *invalid*, que indica el texto para la opción si corresponde a otro mutante diferente al presentado en pantalla. De forma similar a la del DSL *idModel*, cada elemento de tipo *Text* contiene elementos de tipo *Word*, que pueden ser cadenas de texto constantes *Constant*, o palabras clave *Variable*, para indicar qué objeto de la instrucción de mutación se quiere utilizar en el texto.

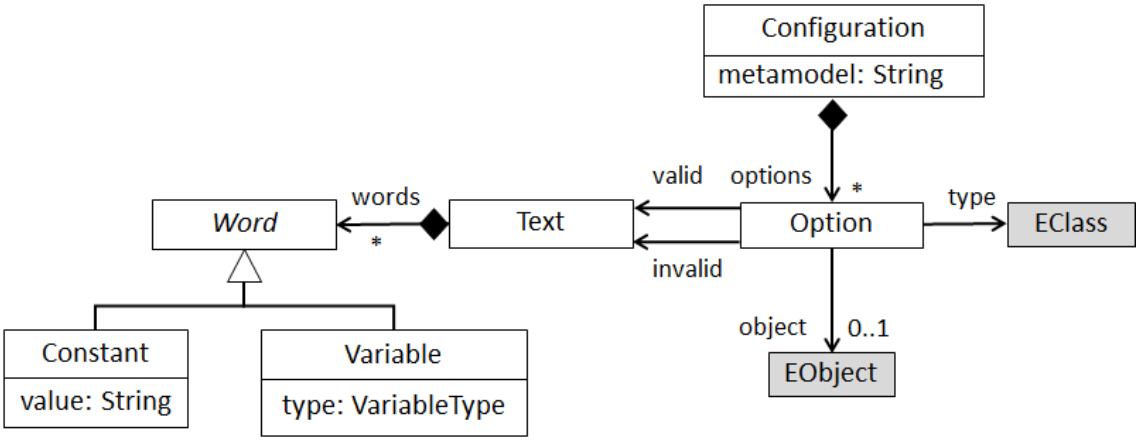

Figura 5.6 Meta-modelo del DSL *configOptions*:
configuración de las opciones en los ejercicios complejos

En el Listado 5.4 se muestra un ejemplo de código del DSL *configOptions*. Este DSL permite indicar el texto específico para cada mutación aplicada (líneas 3 y 6), cuando la opción es una solución (líneas 4 y 7), y cuando no lo es (líneas 5 y 8). *configOptions* permite referirse a los elementos implicados en las mutaciones que se utilizan en el ejercicio mediante una serie de palabras clave (*VariableType*), por ejemplo *%object* sirve para referirse al elemento que se ha mutado, *%refName* es el nombre de la referencia que se utiliza en la mutación, etc (líneas 4-5 y 7-8). Estos parámetros variables se sustituyen posteriormente en el generador de código html, de forma que son también independientes del meta-modelo del dominio. Cuando el parámetro corresponde a un objeto, se utiliza el valor especificado en el DSL *idModel* para expresarlo dentro de la opción en la aplicación web. El símbolo '/' se utiliza para separar la plantilla de texto que se emplea cuando la opción es correcta, que se muestra antes de este símbolo, de la que corresponde a la opción incorrecta, que se muestra después. En este formato de ejercicios, las opciones correctas indican lo que se debe hacer para deshacer las mutaciones aplicadas, que forman parte del mutante presentado en pantalla (líneas 4 y 7), y las opciones incorrectas indican lo que se debe hacer para aplicar las mutaciones de los modelos que mutan el modelo presentado en pantalla (líneas 5 y 8). El DSL *configOptions* tiene además en cuenta el meta-modelo

del dominio (línea 1) para posibilitar la especificación de una configuración de opciones sobre un determinado elemento del modelo. Por ejemplo, AttributeChanged*(State)* (línea 6) especifica que las opciones corresponden a cambios de atributos en elementos de tipo *State*. En este listado, la palabra clave *%object* (líneas 4, 5, 7 y 8), se sustituye por el texto de identificación definido en el DS*L idModel* del elemento que está siendo mutado, o bien, se sustituye por el texto por defecto; %fromObject (líneas 4 y 5), se sustituye por el texto de identificación definido en el DSL *idModel* para el elemento origen del elemento mutado por la instrucción *TargetReferenceChanged* (línea 3), o bien, por el texto por defecto; %refName (líneas 4 y 5), se sustituye por el texto de identificación definido en el DSL *idModel* para la referencia destino del elemento mutado, o bien, por el texto por defecto; etc.

```
1. metamodel "http://fa.com"
2.
3. >TargetReferenceChanged:
4. Change %object from %fromObject to %toObject with new %refName %oldToObject /
5. Change %object from %fromObject to %oldToObject with new %refName %toObject
6. >AttributeChanged(State):
7. Change attribute %attName from %object with value %newValue to %oldValue /
8. Change attribute %attName from %object with value %oldValue to %newValue
```

Listado 5.4 DSL *configOptions*: configuración de las opciones en los ejercicios complejos

```
1. Change Transition from q0 to q1 with new tar q2
2. Change Transition a from State q0 to State q1 with new target q2
3. Change attribute isInitial from q0 with value true to false
4. Change attribute isInitial from State q0 with value true to false
```

Listado 5.5 Opciones de texto generadas:

utilizando el DSL *idModel* en el DSL *configOptions* (líneas 2 y 4), o no, líneas (1 y 3).

A partir de los DSLs *eduTest* y *modelDraw*, Wodel-Edu genera la aplicación web de ejercicios que se corrigen de forma automática, utilizando además de forma opcional los DSLs *idModel* y *configOptions* para la generación de las opciones de los ejercicios complejos. Wodel-Edu genera textos por defecto para identificar los elementos de los modelos, y para los textos que se muestran en los ejercicios complejos en el caso de no implementar configuraciones para los DSLs *idModel* y *configOptions*; El Listado 5.5 muestra las opciones de texto que se generan: la línea 1 muestra el texto por defecto para la mutación de cambio del destino de una transición entre estados: `Change Transition from q0 to q1 with new tar q2`; y la línea 2 muestra el texto generado utilizando los DSLs *idModel* y *configOptions* para la misma instrucción: `Change Transition a from State q0 to State q1 with new target q2`. `Transition a` se genera debido al lenguaje *idModel* (línea 4 del Listado 5.3); `target` es el identificador de la referencia *Transition.tar* (línea 5 del Listado 5.3); etc. El texto utilizado en las instrucciones de cambio de valores de los atributos, la versión que utiliza el DSL el idModel (línea 4) incluye el texto `State q0`, mientras la versión por defecto muestra sólo `q0` para referirse al elemento sobre el que se aplica la mutación. Dependiendo del dominio, el usuario ha de decidir si quiere configurarlos.

5.3. Generación de ejercicios

Se ha generado una aplicación web de ejercicios que sirve como ejemplo y está accesible en la siguiente dirección: www.wodel.eu/tfm/test.html.

El Listado 5.6 muestra el programa Wodel que genera los modelos mutantes que se utilizan en esta aplicación web de ejercicios.

El primer bloque (líneas 5-7) genera 2 mutantes (línea 7) por cada modelo semilla. Estos mutantes se utilizan posteriormente en una pantalla de ejercicios sencillos, en la que el estudiante ha de adivinar cuales son los modelos correctos de entre varios (se muestra el modelo semilla mezclado con estos dos mutantes).

```
1. generate mutants in "out/" from "model/"
2. metamodel "http://fa.com"
3.
4. with blocks {
5.        simpleShowall {
6.                modify target tar from one Transition to other State
7.        } [2]
8.        simpleShowone {
9.                s0 = select one State where {isInitial = true}
10.               s1 = select one State where {isFinal = false}
11.               t0 = select one Transition where {src = s0}
12.               modify one Transition where {tar = s1} with {swapref(tar, t0.tar)}
13.       } [1]
14.       first {
15.               modify target tar from one Transition to other State
16.               modify one State with {reverse(isFinal)}
17.       } [3]
18.       second from first repeat=no{
19.               modify target tar from one Transition to other State
20.               modify one State with {reverse(isFinal)}
21.       } [3]
22.       third from first repeat=no {
23.               modify target tar from one Transition to other State
24.               modify one State with {reverse(isFinal)}
25.       } [3]
26. }
27. constraints {
28.       context State connected: "isInitial or Set{self}->
29.               closure(s | Transition.allInstances()->select(t | t.tar=s)->
30.               collect(src))->exists(s | s.isInitial)"
31. }
```

Listado 5.6 Programa Wodel utilizado para generar los mutantes de la aplicación web evaluada

El segundo bloque (líneas 8-13) genera 1 mutante (línea 13) por cada modelo semilla. Este mutante se utiliza posteriormente en una pantalla de ejercicios sencillos, en la que el estudiante ha de adivinar si los modelos son correctos, o no (se muestra de forma aleatoria el mutante, incorrecto, o el modelo semilla, correcto).

Los bloques tercero, cuarto, y quinto (líneas 14-25) generan los mutantes necesarios para una pantalla de ejercicios complejos, en la que se muestran modelos mutantes junto a opciones de texto seleccionables entre las que el estudiante ha de adivinar cuáles son las soluciones. El bloque tercero (líneas 14-17) genera los

mutantes que se presentan al estudiante en pantalla, y que sirven para generar el texto de las opciones correctas. Las secciones cuarta y quinta (líneas 18-25) generan los mutantes a partir de los que se generan las opciones incorrectas.

Se puede observar que las instrucciones de mutación aplicadas en los bloques tercero, cuarto, y quinto son las mismas: cambio del estado destino de una transición aleatoria (líneas 15, 19, y 24); e inversión del valor del atributo de un estado aleatorio que indica si es final (líneas 16, 20, y 25). El objetivo de hacer esto es generar opciones que sean parecidas entre sí, de forma que no le resulte trivial al estudiante diferenciar las opciones correctas de las que no lo son.

La corrección de los ejercicios se realiza de la siguiente forma, según los tres tipos de ejercicios disponibles:

1. En los ejercicios sencillos en los que se muestran varios diagramas entre los que el estudiante ha de decidir cuál es el correcto, Wodel-Edu comprueba cuál de los diagramas es el que corresponde al modelo semilla a partir del que se han generado los mutantes, y este diagrama es la solución. La Figura 5.7 muestra una captura de pantalla de la aplicación generada con un ejercicio en este formato.

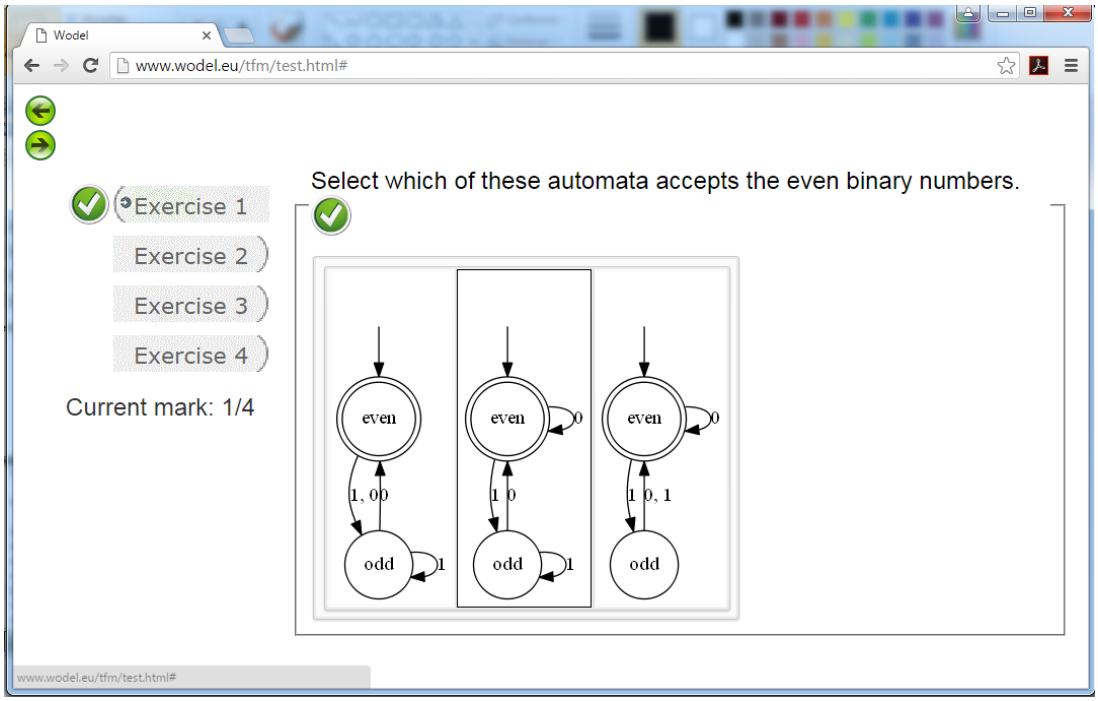

Figura 5.7 Captura de pantalla de la aplicación generada para el 1º formato de ejercicios

2. En el caso de los ejercicios sencillos en los que se muestra un único diagrama y el estudiante ha de decidir si el diagrama es correcto o no, el programa verifica que el diagrama presentado en pantalla corresponde al modelo semilla, caso en el que la solución es que el diagrama es correcto. Si el diagrama presentado no corresponde al modelo semilla, la solución es que el diagrama no es correcto. La Figura 5.8 muestra una captura de pantalla de la aplicación generada con un ejercicio en este formato.

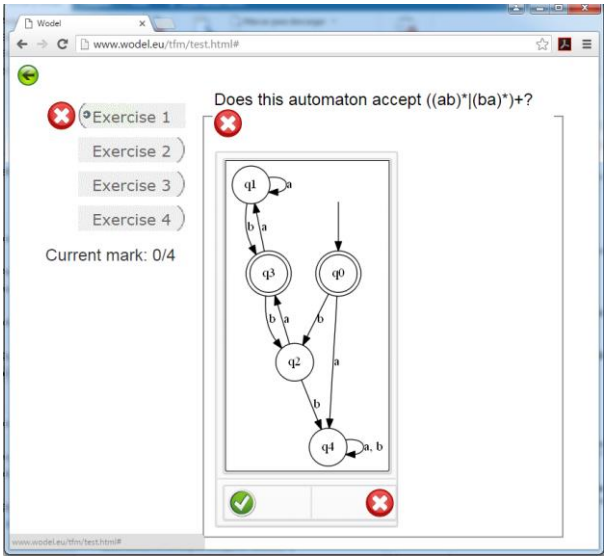

Figura 5.8 Captura de pantalla de la aplicación generada para el 2º formato de ejercicios

3. En el caso de los ejercicios complejos, Wodel-Edu genera a partir del registro de mutaciones las opciones que corresponden al mutante presentado en pantalla (opciones correctas), y las mezcla con otras opciones generadas a partir del histórico de mutantes que tienen como modelo semilla el mutante presentado en pantalla (opciones erróneas). La solución al ejercicio es la respuesta en la que se han marcado sólo las opciones correctas. La Figura 5.9 muestra una captura de pantalla de la aplicación generada con un ejercicio en este formato.

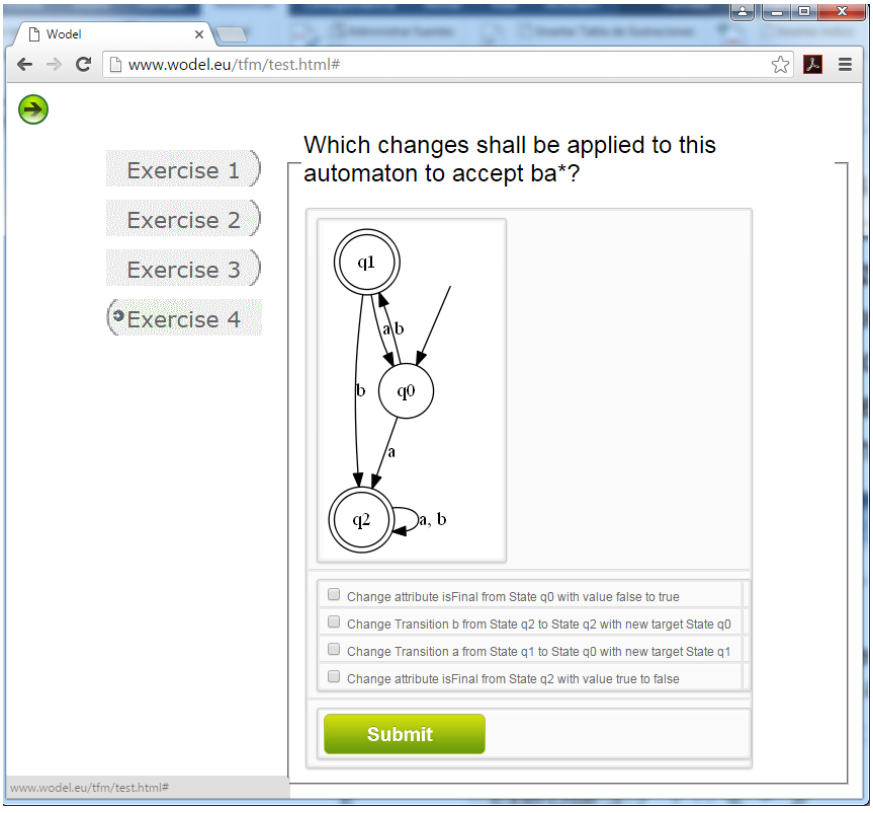

Figura 5.9 Captura de pantalla de la aplicación generada para el 3º formato de ejercicios

A continuación, se describe el proceso realizado por Wodel-Edu para generar el ejercicio presentado en la Figura 5.9. Este ejercicio se ha generado a partir de 3 modelos: el modelo semilla; y otros dos modelos mutantes generados a partir de este modelo. Se parte del modelo semilla original (Figura 5.10), que acepta el lenguaje ba*.

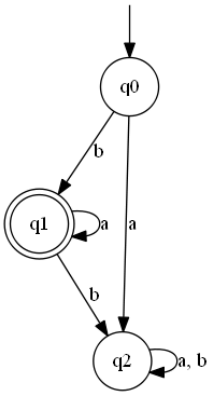

Figura 5.10 Autómata que acepta el lenguaje ba*

A este modelo semilla se le aplican dos instrucciones de mutación para generar el modelo mutante presentado como diagrama en el ejercicio.

1. Cambiar el estado q2 a estado final.
2. Cambiar la transición ‘a’ que va desde el estado q1 hasta el estado q1 con nuevo destino el estado q0.

La Figura 5.11 muestra el modelo mutante generado tras aplicar estas dos instrucciones de mutación sobre el modelo semilla original.

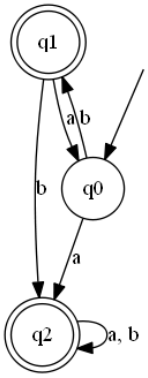

Figura 5.11 Autómata mutante presentado en el ejercicio

Utilizando el registro de las operaciones de mutación aplicadas, se generan las opciones correctas (soluciones) que se presentan en pantalla (ver Figura 5.9). Estas opciones se muestran en pantalla para poder ‘deshacerse’ sobre el modelo presentado, es decir, para recuperar el modelo semilla original que acepta el lenguaje ba*:

1. *Change attribute isFinal from State q2 with value true to false.*

2. *Change Transition a from State q1 to State q0 with new target State q1.*

El siguiente paso es mutar el mutante de la Figura 5.11, de forma que se generan las opciones incorrectas que se presentan en pantalla (ver Figura 5.9).

1. Cambiar el estado q0 a estado final.
2. Cambiar la transición 'b' que va desde el estado q2 hasta el estado q2 con nuevo destino el estado q0.

La Figura 5.12 muestra el mutante resultante utilizado para generar las opciones erróneas.

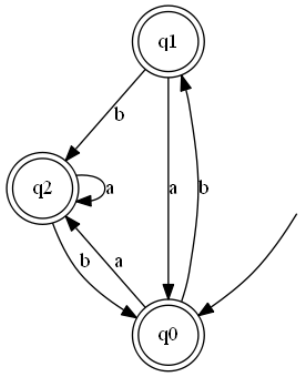

Figura 5.12 Autómata mutante utilizado para generar las opciones erróneas

Se generan las opciones incorrectas (fallos) que se presentan en pantalla (ver Figura 5.9) a partir del registro de mutaciones. Estas opciones se muestran en pantalla para poder aplicarse sobre el modelo presentado, es decir, para crear un modelo diferente (erróneo) del modelo semilla.

1. *Change attribute isFinal from State q0 with value false to true.*
2. *Change Transition b from State q2 to State q2 with new target State q0.*

Una vez se han generado todas las opciones correctas e incorrectas, Wodel-Edu las mezcla entre sí y se presentan en el ejercicio en un orden aleatorio.

5.4. Herramienta proporcionada

Wodel-Edu se integra como extensión en el asistente de creación de proyectos de Wodel. Como se muestra en la Figura 5.13, se pide al usuario que introduzca el nombre del proyecto; el nombre del fichero .mutator del DSL Wodel para la generación de los mutantes; la carpeta en la que se incluyen los modelos semilla; la carpeta donde almacenar los modelos mutantes que se generan; y, por último, un listado con las extensiones de Wodel incluidas. En este caso, se selecciona aquí que se quiere crear un proyecto de Wodel-Edu.

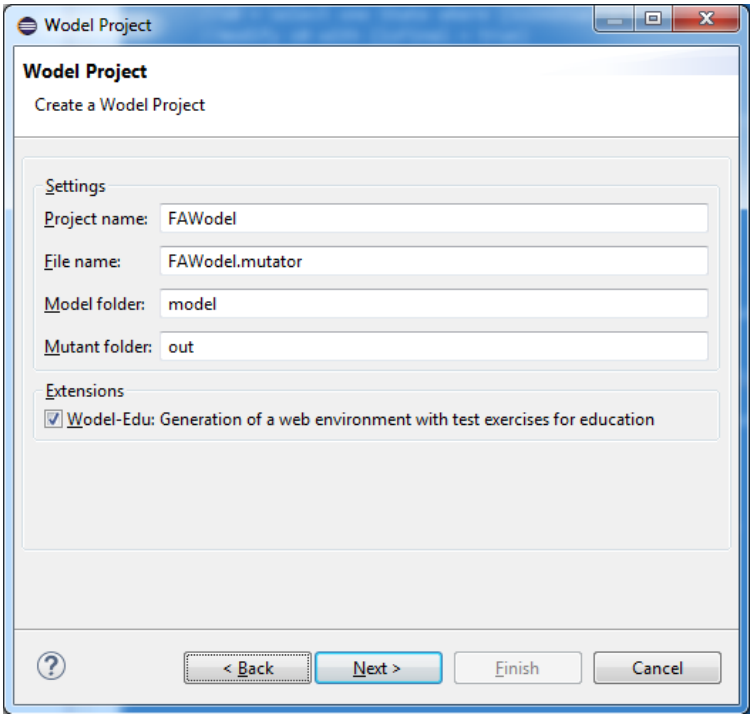

Figura 5.13 Página de configuración del asistente para crear un proyecto de Wodel

A continuación, se pide al usuario que introduzca el meta-modelo del dominio de la aplicación, que se utiliza en los DSLs de Wodel y Wodel-Edu para el completado de código, la validación, y la comprobación de tipos. Los ficheros de código de los DSL se crean con una plantilla del código que sirve de ejemplo al usuario cuando crea un proyecto de Wodel-Edu.

Las Figuras 5.14 y 5.15 muestran dos capturas de pantalla de los editores que se crean, la Figura 5.14 muestra el editor del DSL *modelGraph*, y la Figura 5.15 muestra el editor del DSL *idModel*. En ambos casos se muestra el asistente de completado de código, que presenta diferentes alternativas que se pueden introducir en el código, dependiendo dónde esté escribiendo el usuario.

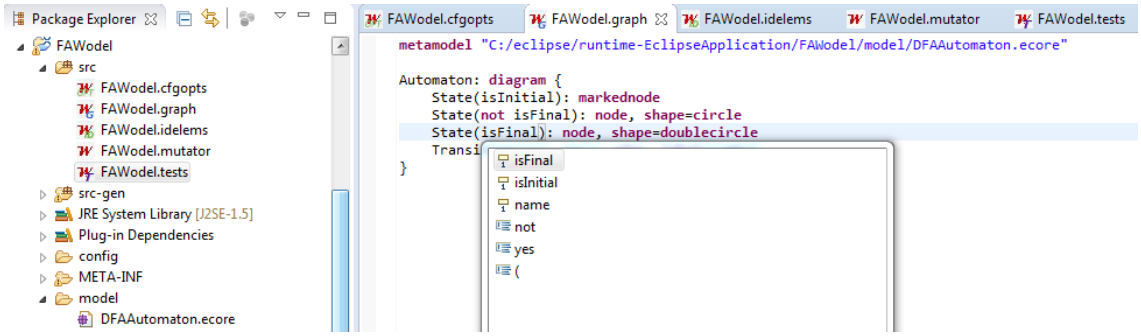

Figura 5.14 Captura de pantalla del editor para el DSL *modelGraph*

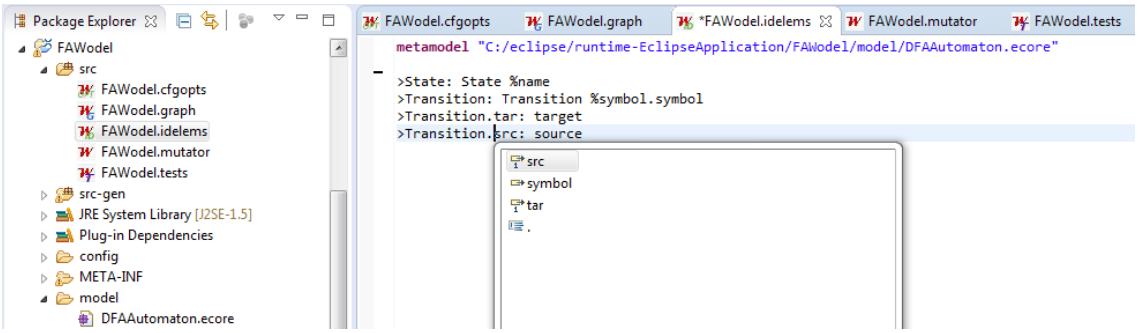

Figura 5.15 Captura de pantalla del editor para el DSL *idModel*

La Figura 5.16 muestra la pantalla de preferencias del editor de Wodel, en la que se puede configurar la generación del registro de mutaciones; la generación de modelos en formato .json; y la compactación del registro. Además, se permite configurar el número máximo de intentos que realizará la aplicación hasta generar un modelo mutante válido, conforme al meta-modelo y que no esté repetido; y también configurar el número máximo de mutantes que se generarán por defecto.

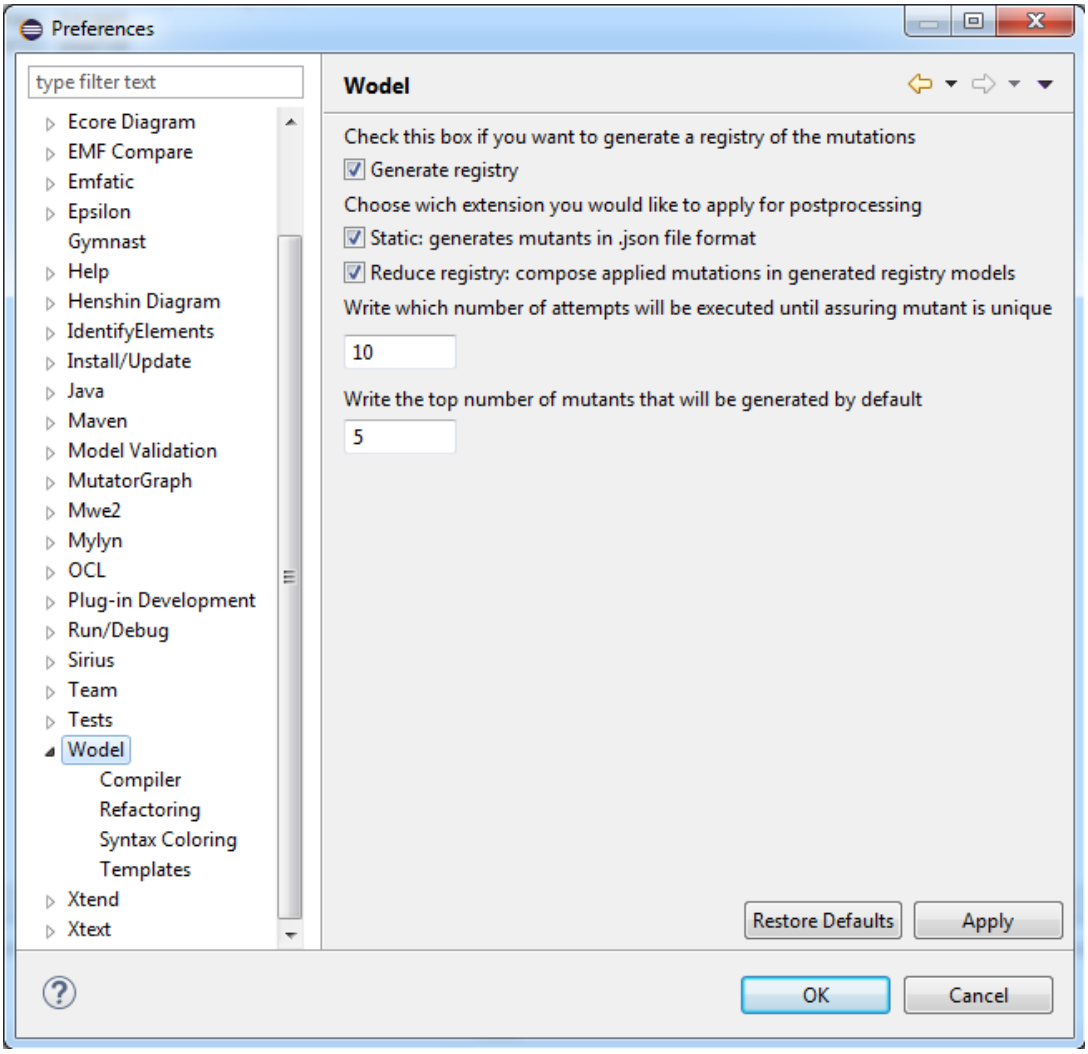

Figura 5.16 Pantalla de preferencias del editor de Wodel

5.5. Evaluación de la calidad de los ejercicios generados

A continuación, se presenta una evaluación preliminar de Wodel-Edu realizada con el objetivo de medir la calidad de los ejercicios que genera. Para ello, se ha utilizado Wodel-Edu para generar una aplicación web en la que, según se explica en la Sección 5.2: la primera página corresponde a los ejercicios complejos con opciones de texto seleccionables; la segunda página corresponde a los ejercicios sencillos en los que hay que seleccionar un diagrama entre varios; y la tercera página corresponde a los ejercicios sencillos en los que hay que decidir si el diagrama es correcto, o no.

La evaluación se ha realizado por 10 usuarios con las siguientes características: sólo 1 no tiene formación en teoría de autómatas; 8 son hombres, y 2 son mujeres; la edad de los participantes está comprendida entre los 22 y los 41 años, siendo la media de edad de los participantes de 31 años. Se pregunta a los usuarios por las siguientes cuestiones, puntuables con valores entre 1 y 5:

- El ejercicio se entiende bien.
- La dificultad del ejercicio es adecuada.
- El ejercicio es útil para aprender autómatas.

Se ha incluido también un campo de texto libre para comentarios o sugerencias sobre cada página de ejercicios. Varios usuarios indican que la interfaz es mejorable, y que les cuesta entender el enunciado, p. ej., que habría que incluir la expresión regular del enunciado entre comillas, o que habría que especificar en la primera página de ejercicios que las opciones que hay que marcar son para que el autómata sólo acepte las palabras del lenguaje indicado en la expresión regular, y ninguna otra palabra.

Un usuario encuentra en el ejercicio 2 de la primera pantalla una opción ‘trampa’, es decir, una opción que no se puede aplicar sobre el autómata presentado en pantalla, como se muestra en la Figura 5.17. Este detalle indica un error en la implementación del generador de la aplicación web, en la generación de las opciones incorrectas. Este error, que era difícil de identificar porque sólo aparecía algunas veces, ha sido corregido posteriormente a la evaluación.

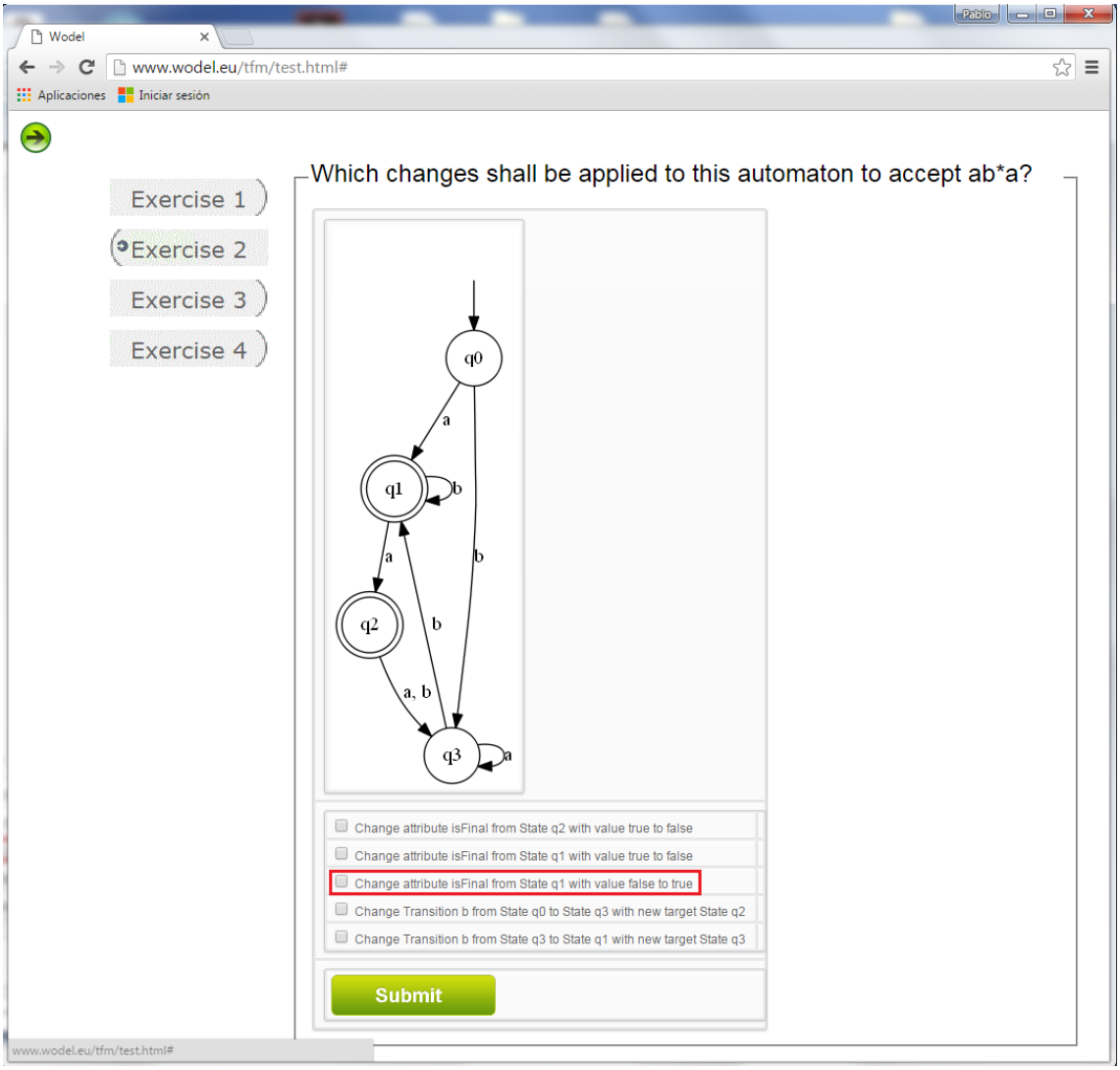

Figura 5.17 Opción 'trampa' en el ejercicio 2 de la primera pantalla de ejercicios

También se incluye en el cuestionario un campo opcional en el que el usuario puede indicar la nota obtenida. De esta forma se puede deducir si los ejercicios tienen una dificultad adecuada. Una de las observaciones indicadas por un usuario es que en la inversión del atributo que indica que el estado es o no final, quedaría más legible mostrar *'make State q1 final'*, o *'make State q1 non final'*, cuestión que queda como trabajo futuro.

En las Figuras 5.18, 5.19, y 5.20 se muestra la media de las valoraciones de las tres dimensiones que se han tenido en cuenta: legibilidad, dificultad, y utilidad para el aprendizaje de autómatas. En la Figura 5.18 se observa que la inteligibilidad de los ejercicios es mejorable, obteniendo la primera página de ejercicios la valoración media más baja, de un 68%, siendo la segunda del 78% y del 76% la tercera. No obstante, la nota más baja (1 sobre 5) es otorgada por el participante que no tiene formación en teoría de autómatas En cuanto a la dificultad, en la Figura 5.19 se observa que la primera página de ejercicios resulta también la más complicada, obteniendo una valoración media del 82%, el 89% la segunda y el 86% la tercera. En la Figura 5.20 se observa que la valoración sobre la utilidad de los ejercicios es alta, siendo la utilidad

más baja la de la tercera página de ejercicios con una valoración media del 88%, y un 94% la segunda y tercera.

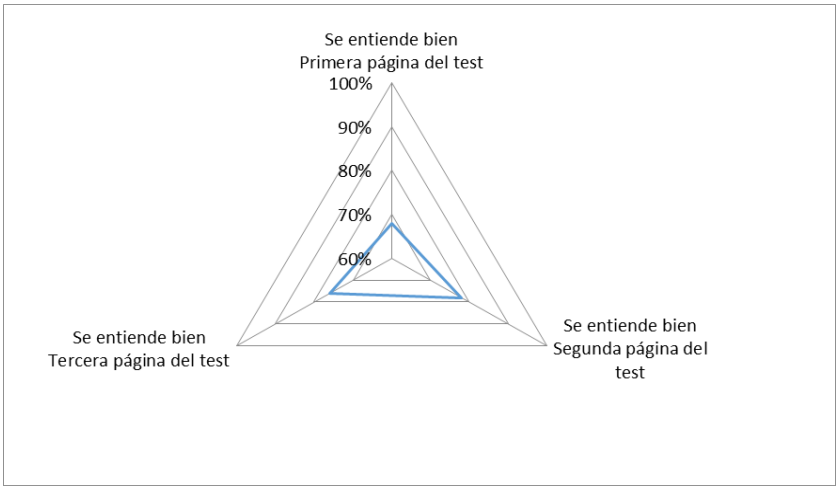

Figura 5.18 Media de la legibilidad de las tres páginas de ejercicios

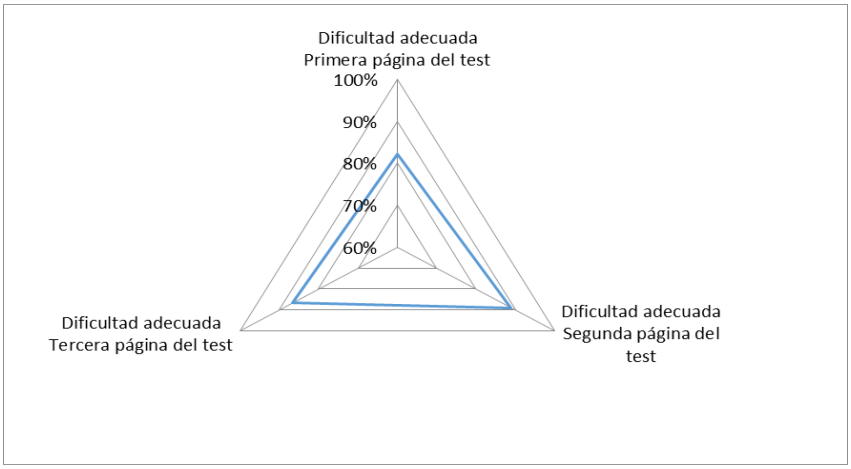

Figura 5.19 Media de la dificultad de las tres páginas de ejercicios

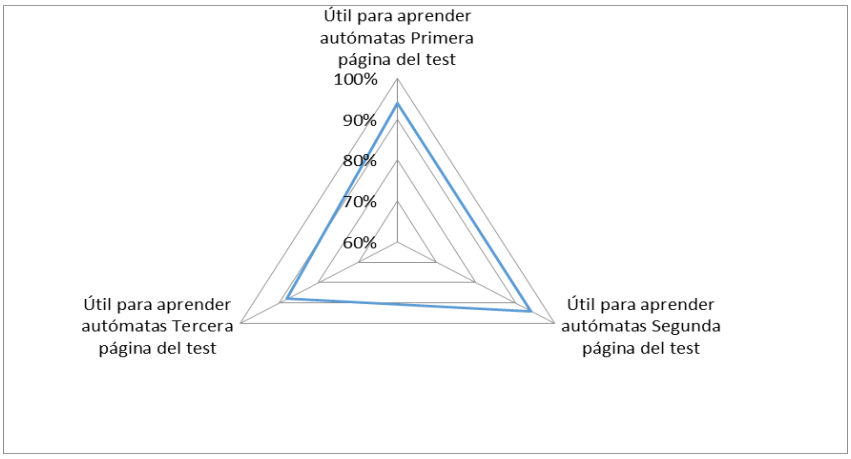

Figura 5.20 Media de utilidad para el aprendizaje de autómatas de las tres páginas de ejercicios

En la Figura 5.21 se muestra la media de las notas obtenidas, aunque sólo un 60% de los usuarios incluyó su calificación en el cuestionario. Se puede observar que la calificación media es superior al 60% para el caso de las segunda y tercera páginas del test, siendo la calificación media de la primera página de un 50%. Dos participantes obtuvieron el 100% de la calificación posible en la primera página, uno en la segunda y dos en la tercera. Ningún participante obtuvo el 100% en todas las páginas del test.

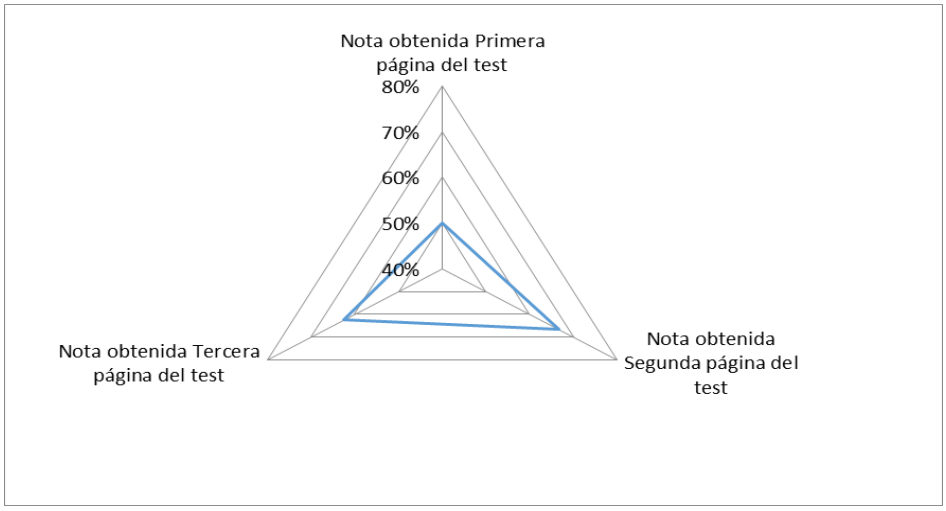

Figura 5.21 Media de la calificación obtenida

Se pueden extraer varias conclusiones interesantes de esta evaluación preliminar. Por los resultados se deduce que el ejercicio de seleccionar opciones (página 1) es complicado de entender. Se puede hacer más sencillo utilizando la opción de agrupación de opciones según el mutante con el parámetro de configuración *mode=radiobutton* (ver Listado 5.1), o haciendo que el ejercicio sea interactivo, permitiendo que el usuario modifique el autómata para obtener la solución correcta. También parece deducirse que los ejercicios tienen una dificultad percibida razonable, y que se perciben como muy útiles para el aprendizaje de autómatas. Otra cuestión importante es la de incluir instrucciones para realizar los ejercicios, con un tutorial o similar. Todas estas ideas resultan muy útiles para evaluaciones posteriores y mejoras de la herramienta.

En cuanto a las amenazas a la validez del experimento, podemos indicar que el orden de presentación de los ejercicios ha podido influir en la percepción de su complejidad. Por ejemplo, se han puesto los ejercicios más difíciles al principio, lo que puede influir en la dificultad percibida de la segunda hoja de ejercicios. De manera adicional, sólo se han evaluado ejercicios generados automáticamente. Se podrían mezclar ejercicios generados de forma automática con otros hechos a mano, de forma que se puedan comparar las valoraciones que obtienen unos y otros. También hay que tener en cuenta que los usuarios participantes en la evaluación no son estudiantes reales de una asignatura de autómatas. Finalmente, sería necesaria una evaluación de la herramienta desde el punto de vista del profesor que diseña los ejercicios.

Capítulo 6. Conclusiones y trabajo futuro

En este trabajo se ha presentado Wodel, un DSL para especificar operadores de mutación y programas de mutación independientes del dominio, cuyo entorno de desarrollo puede extenderse para aplicaciones diferentes en dominios diferentes. Se ha presentado Wodel-Edu, que es una aplicación de Wodel en el dominio de la educación, y consiste en un entorno de generación automática de ejercicios mediante técnicas de mutación implementado como extensión de Wodel. Estos ejercicios pueden ser a su vez de dominios diferentes. En este trabajo se ha utilizado Wodel-Edu para generar ejercicios de autómatas finitos, pero se puede utilizar para generar ejercicios de otros dominios, p. ej., diagramas de clases UML, circuitos electrónicos, etc.

De la evaluación preliminar se puede concluir que los ejercicios resultan útiles a los usuarios para el aprendizaje de autómatas, como se puede ver en la Figura 5.20. Sin embargo, en la Figura 5.18 se observa también que podría mejorarse la legibilidad de los ejercicios, especialmente los de selección de opciones. Esta evaluación preliminar nos da una idea de cómo plantear mejor posteriores evaluaciones o usos reales de los ejercicios generados: ordenando los ejercicios de más fáciles a más difíciles; agrupando las opciones en los ejercicios complejos; comparando la evaluación de ejercicios generados con Wodel-Edu y otros hechos a mano; e incluyendo las instrucciones para que el estudiante tenga claro el problema que tiene que resolver cuando se enfrenta a una pantalla de ejercicios.

Como trabajo futuro, pensamos mejorar la legibilidad del texto de las opciones de los ejercicios. También nos proponemos generar otros formatos de diagramas utilizando el DSL *modelDraw* de Wodel-Edu. Otro trabajo futuro es utilizar Wodel-Edu para generación de ejercicios de otros dominios (p. ej., circuitos electrónicos, diagramas de clases UML, etc.). Otro trabajo pendiente es crear librerías de ejercicios, con instrucciones de mutación interesantes, y configuraciones para el Wodel-Edu, para estos dominios, que se puedan cargar de forma automática mediante un asistente al crear un proyecto de Wodel-Edu. También nos planteamos crear una librería Java para integrar fácilmente este entorno en aplicaciones web. Llevaremos a cabo pruebas reales de la herramienta en el ámbito educativo (por ejemplo, en una asignatura de la titulación de Ingeniería Informática). Mejoraremos los lenguajes del plug-in Wodel-Edu para ofrecer entornos de aprendizaje más complejos (p. ej., que incluyan gamificación), y ejercicios (p. ej., ejercicios interactivos donde el estudiante modifique el diagrama mutado). También otro objetivo de trabajo futuro es que estos

ejercicios puedan realizarse desde terminales móviles, y aprovechar de esta forma el reciente trabajo del grupo MISO sobre modelado de dominio específico en dispositivos móviles [32]. Otra idea interesante es la de generar mutantes que no sean conformes al meta-modelo del modelo semilla (por ejemplo, que violen una cardinalidad o una restricción OCL).

También pensamos extender el lenguaje Wodel a otros dominios, por ejemplo, las pruebas basadas en modelos, algoritmos genéticos, o la generación de modelos grandes. La aplicación de técnicas de mutación puede resultar útil para áreas tan diversas como la composición musical [21].

Bibliografía

[1] Aichernig, B. K., Brandl, H., Jöbstl, E., and Krenn, W. *UML in action: A two-layered interpretation for testing*. SIGSOFT Softw. Eng. Notes 36, 1 (2011), 1-8.

[2] Aranega, V., Mottu, J., Etien, A., Degueule, T., Baudry, B., and Dekeyser, J. *Towards an automation of the mutation analysis dedicated to model transformation*. Softw. Test., Verif. Reliab. 25, 5-7 (2015), 653-683.

[3] Bartel, A., Baudry, B., Munoz, F., Klein, J., Mouelhi, T., and Traon, Y. L. *Model driven mutation applied to adaptative systems testing*. In ICST Workshops (2011), pp. 408-413.

[4] Bettini, L. *Implementing Domain-Specific Languages with Xtext and Xtend* (2013), Packt Publishing. https://eclipse.org/Xtext/ https://eclipse.org/xtend/

[5] Blumenstein, M., Green, S., Nguyen, A. and Muthukkumarasamy, V. *An experimental analysis of GAME: a generic automated marking environment*. In Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education, pp 67-71 (2004).

[6] Bombieri, N., Fummi, F., Guarnieri, V., and Pravadelli, G. *Testbench qualification of SystemC TLM protocols through mutation analysis*. IEEE Trans. Comput. 63, 5 (2014), 1248-1261.

[7] Brambilla, M., Cabot, J., and Wimmer, M. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, USA, (2012).

[8] Clayberg, E. and Rubel, D. *Eclipse: Building Commercial-Quality Plug-Ins (2nd Edition) (Eclipse).* Addison-Wesley Professional (2006).

[9] Cordy, M., Classen, A., Perrouin, G., Schobbens, P. Y., Heymans, P. and Legay, A., *Simulation-based abstractions for software product-line model checking*, 2012 34th International Conference on Software Engineering (ICSE), pp. 672-682 Zurich (2012).

[10] Devroey, X., Perrouin, G., Schobbens, P.-Y., and Heymans, P. *Poster: Vibes, transition system mutation made easy*. In Software Engineering (ICSE), (2015) IEEE/ACM 37th IEEE International Conference on (2015), vol. 2, pp. 817-818.

[11] Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag New York, Inc (2006).

[12] *Free Problem Set (FPS)*: http://code.google.com/p/freeproblemset/ (2010).

[13] Gómez-Abajo, P., de Lara, J., Guerra, E. *Wodel: A Domain-Specific Language for Model Mutation*. (2016) SAC'2016, (Pisa) (ACM). pp.:1-6.

[14] He, X., *A metamodel for the notation of graphical modeling languages*. Computer Software and Applications Conference, COMPSAC 2007 - Vol. 1. 31st Annual International, Volume 1, Issue, 24–27, pp 219-224 (2007).

[15] Henard, C., Papadakis, M., and Traon, Y. L. *Mutalog: A tool for mutating logic formulas*. In ICST Workshops Proceedings (2014), IEEE CS, pp. 399-404.

[16] Lackner, H., and Schmidt, M. *Towards the assessment of software product line tests: A mutation system for variable systems*. In Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2 (New York, NY, USA, 2014), SPLC '14, ACM, pp. 62-69.

[17] Mandal, A.K., Mandal, C. and Reade, C.M.P. *Architecture of an Automatic Program Evaluation System*. In CSIE Proceedings (2006).

[18] Mernik, M., Heering, J., Sloane, A. M*. When and how to develop domain-specific languages*. ACM Computing Surveys, 37(4):316–344, (2005).

[19] Moawad, A., Hartmann, T., Fouquet, F., Nain, G., Klein, J., and Bourcier, J. Polymer – *A model-driven approach for simpler, safer, and evolutive multi-objective optimization development*. In MODELSWARD (2015), SciTePress, pp. 286-293

[20] Nguyen, P. H., Papadakis, M., and Rubab, I. *Testing delegation policy enforcement via mutation analysis*. In ICST Workshops (2013), pp. 34-42.

[21] Ortega, A., Sánchez Alfonso, R., Alfonseca, M. *Automatic composition of music by means of grammatical evolution*. APL 2002: 148-155

[22] Pietsch, P., Yazdi, H. S., and Kelter, U. *Controlled generation of models with defined properties*. In Software Engineering (2012), vol. 198 of LNI, GI, pp. 95-106.

[23] Queirós, R. A. and Leal, J. P. *PETCHA: a programming exercises teaching assistant. In Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education (ITiCSE '12)*. ACM, New York, NY, USA, 192-197. (2012).

[24] Rodrigues da Silva, A., *Model-driven engineering: A survey supported by the unified conceptual model*, Computer Languages, Systems & Structures, Volume 43, Pages 139-155 (2015).

[25] Sadigh, D., Seshia, S. A., and Gupta, M. *Automating exercise generation: A step towards meeting the MOOC challenge for embedded systems*. In WESE (2013), ACM, pp. 2:1-2:8.

[26] Silberschatz, A., Korth, H., and Sudarshan, S. *Database Systems Concepts (5 ed.)*. McGraw-Hill, Inc., New York, USA (2005).

[27] Simao, A. S., and Maldonado, J. C. *MuDeL: a language and a system for describing and generating mutants*. J. Braz. Comp. Soc. 8, 1 (2002), 73-86.

[28] Soler, J., Boada, I., Prados, F., Poch, J. and Fabregat, R., *A web-based e-learning tool for UML class diagrams*, IEEE EDUCON 2010 Conference, Madrid, pp. 973-979 (2010).

[29] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. *EMF: Eclipse Modeling Framework 2.0 (2nd ed.) (2009).* Addison-Wesley Professional.

[30] Stephan, M., Alalfi, M. H., Stevenson, A., and Cordy, J. R. *Using mutation analysis for a model-clone detector comparison framework*. In ICSE (2013), IEEE / ACM, pp. 1261-1264.

[31] Traetteberg, H, Aalberg, T. *JExercise: A specification-based and test-driven exercise support plug-in for Eclipse*. In Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange, 70-74, ETX 2006 (2006).

[32] Vaquero-Melchor, D., Garmendía, A., Guerra, E., and de Lara, J., *Towards enabling mobile domain-specific modelling*. (2016) ICSOFT'2016, Lisbon, pp.:1-6.

[33] Verhoeff, T. *Programming Task Packages: Peach Exchange Format*. In Olympiads in Informatics, Vol. 2 192-20 (2008).

[34] Vincenzi, A. M. R., Simao, A. S., Delamaro, M. E., and Maldonado, J. C. *Muta-Pro: Towards the definition of a mutation testing process*. J. Braz. Comp. Soc. 12, 2 (2006), 49-61.

[35] Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E., Wachsmuth, G. *Engineering - Designing, Implementing and Using Domain-Specific Languages*, dslbook.org (2013).

[36] Warmer, K. *The Object Constraint Language 2nd Edition. Getting your Models Ready for MDA*. Addison-Wesley (2003).

[37] http://acme.udg.edu/es/equip.php, sitio web de la plataforma ACME (visitado por última vez en junio de 2016).

[38] https://eclipse.org/atl/, sitio web de ATL (visitado por última vez en junio de 2016).

[39] https://eclipse.org/modeling/emf/, sitio web de EMF (visitado por última vez en junio de 2016).

[40] https://www.eclipse.org/henshin/, sitio web de Henshin (visitado por última vez en junio de 2016).

[41] http://www.omg.org/mof/, sitio web de MOF (visitado por última vez en junio de 2016).

[42] http://www.omg.org/spec/OCL/2.4/, especificación de OCL 2.4 (visitado por última vez en junio de 2016).

[43] http://www.omg.org/spec/QVT/, especificación de QVT (visitado por última vez en junio de 2016).

[44] http://www.uml.org/, sito web de UML(visitado por última vez en junio de 2016).