Mutation Testing for Java based on Model-Driven Development

Ariel Gonzalez
Universidad Nacional de Río Cuarto,
Río Cuarto, Argentina

Carlos Luna
Universidad de la República,
Montevideo, Uruguay

Gonzalo Bressan
Universidad Nacional de Río Cuarto,
Río Cuarto, Argentina

***Abstract*—This article presents an implementation, based on model-driven development, that supports mutation testing techniques for the evaluation of test cases. The mutation of the code is carried out through a process of model transformations that starts with a transformation of a JAVA program to a representation of it in XMI, which satisfies the requirements of the JAVA metamodel. Later, mutation rules are applied to the model by means of a transformation, model by model, to generate a mutated version of the original program. Finally, JAVA code mutated again through a transformation, model to text, defined with MOF2Text. This last version is the one used to experiment with the different test cases. The main contribution of this work is the proposal of a mutation mechanism of JAVA programs that makes use of existing standards and tools in the context of model-driven development.**

***Keywords*– Java, Mutation, Java Models, QVT Operational.**

I. Introducción

El desarrollo dirigido por modelos (Model-Driven Development, MDD) [1] es una metodología de la ingeniería de software (en algunas literaturas es sinónimo de Model-driven Engineering, MDE) que, durante años, ha representado los artefactos de software como modelos, con el objetivo de incrementar la productividad, calidad y reducir los gastos en el proceso de desarrollo de software. Tiene como objetivo organizar los niveles de abstracción y las metodologías de desarrollo, promoviendo el uso de modelos como artefactos principales a ser construidos y mantenidos. Un modelo está constituido por un conjunto de elementos que proporcionan una descripción sintética y abstracta de un sistema, concreto o hipotético. De esta manera, el proceso de desarrollo se convierte en un proceso de refinamiento y transformación entre modelos, de manera que el nivel de abstracción cada vez es menor, hasta que en un último paso se genera código para una plataforma específica. Desde hace algunos años existe un interés creciente en este campo. En particular, la arquitectura dirigida por modelos (Model Driven Architecture, MDA) [2] es la aproximación definada por el Object Management Group (OMG) como una aproximación a la especificación e interoperabilidad de sistemas basada en el uso de modelos formales. En MDA los modelos independientes de la plataforma son inicialmente expresados en un lenguaje de modelado independiente de la plataforma, como UML [3]. El modelo independiente de la plataforma es traducido a un modelo específico para una plataforma considerada; por ejemplo, la plataforma JAVA. Por último, a partir del modelo específico para la plataforma se genera el código del sistema en el lenguaje de programación objetivo (JAVA, C#, etc.).

La prueba por mutación se originó como una técnica propuesta por Richard Lipton mientras era estudiante en el año 1971 [4], y la primer implementación de una herramienta que da soporte al testeo por mutación fue desarrollada por Timothy Budd como parte de su posdoctorado. Recientemente, con la disponibilidad del poder computacional masivo, ha habido un resurgimiento del análisis de mutación dentro de la comunidad informática, y se ha trabajado para definir métodos para aplicar pruebas de mutación tanto a lenguajes de programación como a modelos [5]. La técnica de mutación se basa en dos hipótesis. La primera es conocida como la *Hipótesis del Programador Competente* y sostiene que las fallas que puede cometer un programador experimentado se reducen a meros errores sintácticos. La segunda hipótesis es conocida como *Efecto de Acoplamiento* y sostiene que fallas simples en el código pueden generar otros errores en cascada. Uno de los objetivos de este mecanismo es diseñar nuevas pruebas de software y evaluar la calidad de las existentes. La técnica de mutación basada en código (en adelante nos referimos siempre a mutación sobre código y no sobre modelos) requiere hacer un pequeño cambio en el código fuente. Cada versión cambiada es un mutante y las pruebas deben ser capaces de distinguir el programa original del mutante a partir del comportamiento observado, a esto se le llama "matar al mutante". Un conjunto de pruebas es evaluado en base al porcentaje de los mutantes que mata (puntuación de mutación). Los mutantes sobrevivientes se pueden usar para diseñar nuevas pruebas. Los mutantes son creados utilizando operadores de mutación concretos que imitan errores comunes de programación, por ejemplo, utilizar operadores o variables equivocados, forzar a que se divida por cero, etc. El objetivo es ayudar al desarrollador a crear pruebas efectivas localizando carencias en el conjunto de pruebas, especialmente en los puntos del programa que son menos transitados durante su ejecución.

En este trabajo se implementan operadores de mutación sobre código JAVA en el contexto MDD para dar soporte a la técnica de testing por mutación (Mutation Testing, MT).

A. Contribución

Este trabajo propone generar mutantes mediante la aplicación de transformaciones *Text-To-Model* (T2M), *Model-To-Model* (M2M) y *Model-To-Text* (M2T) de un programa escrito en código JAVA, en el contexto MDA, para poder aplicar la técnica MT, que permite comprobar la calidad de un conjunto de casos de prueba previamente definidos. El proceso propuesto contempla las siguientes etapas:

- Realizar transformaciones T2M de programas escritos

en código JAVA a su representación en XMI, que en adelante denominamos modelo JAVA.

- Aplicar operadores de mutación comunmente usados sobre los modelos mediante transformaciones M2M.
- Generar código JAVA a partir de un modelo JAVA mediante transformaciones M2T.
- Mostrar la utilidad de las pruebas por mutación como herramienta para evaluar casos de prueba.

La Figura 1 muestra gráficamente las etapas del proceso de transformaciones de modelos.

El foco principal del trabajo es la implementación de operadores de mutación a partir de un modelo XMI del código fuente. Esto evita el costo de realizar múltiples compilaciones de mutantes durante su generación.

Las herramientas existentes basan la mutación del código en la generación de un Árbol de Sintaxis Abstracta (Abstract Sintax Tree, AST) mediante la manipulación de algún parser de JAVA. Esto genera complicaciones para la comunidad de ingeniería de software. Este trabajo aborda la mutación a través del uso de herramientas y definiciones de transformación de modelos; área muy conocida y utilizada por los ingenieros de software.

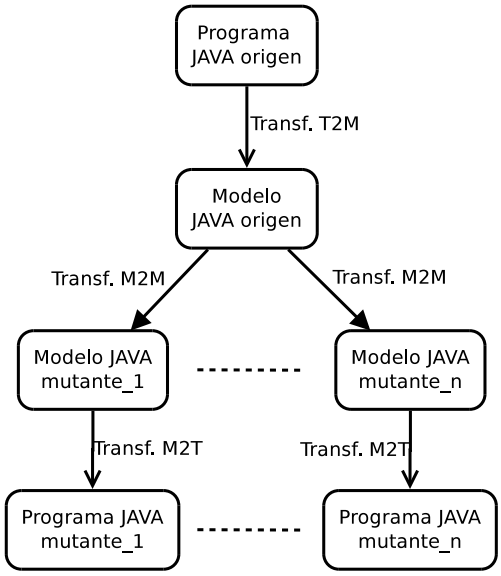

Fig. 1: Mutación basada en transformación de modelos.

B. Organización del Trabajo

El trabajo es organizado de la siguiente manera. En la Sección II se presentan trabajos relacionados con esta investigación. La Sección III introducen metodologías, estándares y la teoría general que constituyen la base de la problemática planteada. En la Sección IV se describen los aspectos más importantes relacionados a la transformación que implementa los operadores de mutación sobre un modelo JAVA. Un ejemplo de aplicación de la propuesta se exhibe en la Sección V. Finalmente, en la Sección VI se discuten las ventajas y prestaciones que ofrece este proceso de transformación de modelos y se presentan algunas actividades relevantes a llevar a cabo próximamente.

II. Trabajos Relacionados

En esta sección se presentan herramientas existentes vinculadas a la mutación de programas. Se analizan además, una serie de trabajos recientes que vinculan las técnicas de MT, sus distintas formas de aplicación y las transformaciones de modelos.

MuJava [6] es un sistema de mutación para programas codificados en lenguaje JAVA. Este sistema automáticamente genera mutantes para MT y Class-Level Mutation Testing. MuJava puede testear clases individuales y paquetes de múltiples clases. En [7], el autor refina y extiende MuJava agregándole funcionalidades como la integración con JUnit, mejoramiento de la API para utilizar la herramienta, incorporación de genericidad, flexibilidad en las opciones de mutación y mutación de referencias, entre otras.

El framework Major [8], [9], [10] posibilita el análisis de mutación de grandes sistemas de software. Esta herramienta provee una fácil detección de fallas e incluye además un lenguaje específico para configurar el proceso de mutación y un analizador para casos de prueba con JUnit.

PIT [11] es una herramienta open source capaz de aplicar pruebas de mutación a programas JAVA de una manera ágil, generando reportes fáciles de comprender que incluyen la cobertura de líneas y de mutación. Es una herramienta muy flexible ya que es acoplable con Maven, Gradle, Ant y otras herramienta de software para la gestión y construcción de proyectos Java.

Jester [12] es un analizador de pruebas basado en JUnit y MT. En particular, ofrece una estrategia para extender el conjunto de operadores de mutación tradicionales. Repite el proceso de crear, ejecutar, probar y reportar para cada mutante durante el análisis; por lo tanto, es muy lento para trabajar. El costo de usar Jester es muy alto en términos de tiempos; el tiempo de máquina y el de interpretar los resultados. Además, no proporciona una interfaz para seleccionar los operadores de mutación aplicados a los programas bajo prueba.

Judy [13], desarrollada por Madeyski y Radyk, es una herramienta de mutación por línea de comandos que soporta la generación y compilación a bytecode y la ejecución de mutantes. Judy implementa operadores tradicionales y operadores de mutación orientados a objetos, y soporta JUnit. Genera mutantes desde código fuente JAVA y evita múltiples compilaciones de mutantes durante su generación.

Javalanche [14] es un framework de código abierto para MT de los programas JAVA con un enfoque especial en la automatización, la eficiencia y la eficacia. Además, manipula bytecode de manera directa para evitar los costos de la recompilación.

Milu [15] es una herramienta eficiente y flexible para aplicar pruebas de mutación sobre programas codificados en lenguaje C. Está diseñada para pruebas de mutación de primer orden y de alto nivel.

Existe una variedad de herramientas adicionales de mutación, [16], [17] entre otras, con objetivos y funciones similares a las anteriores.

En [18] los autores proponen utilizar las técnicas de MT en etapas tempranas del desarrollo de software. Detectan que, como cualquier otro programa, las transformaciones definidas en QVT, ATL u otro lenguaje para tal fin, no están exentas de fallas, las cuales deben ser detectadas a través de pruebas. Este trabajo propone modelos genéricos con fallas que están relacionados con el proceso de transformación del modelo. Los autores identifican las operaciones abstractas que constituyen este proceso: la navegación del modelo, el filtrado de elementos del mismo, la creación del modelo de salida y la modificación del recibido como parámetro de entrada. Luego, proponen un conjunto de operadores de mutación específicos que se inspiran directamente en estas operaciones. Entienden que tales operadores son significativos ya que gran parte de los errores en una transformación se deben a la manipulación de modelos complejos, independientemente del lenguaje de implementación utilizado.

[19] es otro trabajo que enfoca el testing sobre las transformaciones de modelos. Los autores presentan un estado del arte de tests sobre las transformaciones y aducen que es diferente del test de código; por lo tanto surgen nuevos desafios. En particular, analizan la generación de casos de test a través de MT sobre los modelos.

En [20], se propone un mecanismo de refactorización de código JAVA basado en transformación de modelos. Aquí, al igual que en el presente trabajo, los autores construyen el modelo XMI que representa el programa JAVA (modelo JAVA). Luego, mediante transformaciones M2M implementan reglas de refactoreo de código sobre el modelo JAVA y posteriormente obtienen el programa refactorizado de este lenguaje de programación.

El presentel trabajo describe un mecanismo muy particular de aplicación de las técnicas ineherentes a MT, que se basa en el tratamiento de los modelos JAVA (XMI) y no opera sobre el programa fuente. Esto es posible debido a la existencia de nuevas herramientas que permiten obtener dicho modelo. De esta manera se evita, en particular, la necesidad de manipular un parser del lenguaje. Por otro lado, el hecho de construir un modelo del programa fuente y trabajar sobre éste evita el costo de realizar múltiples compilaciones de mutantes durante su generación. Además, a partir del modelo Java obtenido es posible considerar información (metadata) adicional, que puede ser de gran utilidad para la generación de mutantes más complejos.

III. Nociones Preliminares

En esta sección se presentan los principales conceptos referentes a las áreas de base que dan soporte al trabajo: MDD, MT y transformaciones de modelos. La Sección III-A describe y ejemplifica el testeo por mutación. La Sección III-B introduce los principios de MDD; en particular la propuesta concreta del OMG conocida como MDA. Finalmente, la Sección III-C presenta una introducción a las herramientas utilizadas en las distintas transformaciones.

A. Testeo por Mutación (Mutation Testing)

Las pruebas basadas en mutación [21] constituyen un enfoque propuesto por DeMillo, Lipton y Sayward [4] para evaluar la calidad de un conjunto de casos de prueba. La idea consiste en crear ligeras variaciones, llamadas mutantes, del código a probar. Posteriormente, se ejecutan los casos de prueba sobre los mutantes con el fin que éstos detecten el fallo inyectado en el código, comparando la salida esperada con la generada por la ejecución. En caso que difieran se dice que se ha matado al mutante. Cuantos más mutantes se maten mejor es el conjunto de casos de prueba.

Concretamente un mutante es una copia de un programa en la que se ha insertado un pequeño cambio. Este cambio se aplica al código fuente del software y comúnmente en una sola sentencia del código. Los mutantes generados al introducir exactamente un cambio en un programa se conocen como mutantes de primer orden [22]. Los mutantes de segundo orden se generan haciendo dos cambios simples. Los mutantes con más de un cambio también se conocen como mutantes de orden superior. De acuerdo con el *efecto de acoplamiento*, es probable que los mutantes de alto orden sean detectados por los casos de prueba que detectan mutantes de primer orden. Por lo tanto, en general solo se generan mutantes de primer orden y se usan en pruebas de mutaciones.

Se han propuesto diversos tipos de mutaciones, llamados operadores de mutación. Por ejemplo:

- Reemplazo de un operador binario: reemplaza las ocurrencias de operadores binarios tales como operadores aritméticos, lógicos, condicionales y relacionales.
- Sustitución de un operador unario: reemplaza las ocurrencias de operadores unarios como la negación.
- Reemplazo de un valor constante: reemplaza las constantes numéricas y de cadena con constantes predefinidas.
- Manipulación de una condición de rama: manipular las condiciones de rama sin conectores lógicos.
- Eliminación de sentencias: elimina sentencias únicas como llamadas a métodos.

A continuación se expone un ejemplo de reemplazo de un operador binario. Código parcial del programa original:

```
int a = 9;
int x = 5;
a = x + 5;
```

La siguiente es una versión del programa anterior a la cual se le cambia una instrucción (mutante). En este caso, la mutación es representada por un cambio del operador binario aritmético $+$ por $-$:

```
int a = 9;
int x = 5;
a = x - 5;
```

En la literatura actual existe una extensa lista de operadores de mutación, con una identificación que es usual a los miembros de la comunidad de testeo. Algunos de éstos son implementados en este trabajo. Los más comunmente usados son:

- AORU: reemplaza cada ocurrencia del operador unario + por el operador unario - y viceversa.
- COD: elimina cada ocurrencia del operador !.
- COI: inserta el operador ! en expresiones de tipo booleano.
- ROR: sustituye cada ocurrencia de los operadores <, <=, >, >=, == y != por alguno de ellos.
- SOR: reemplaza cada ocurrencia de los operadores «, » y »> por alguno de ellos.

Cada uno de los operadores de mutación se aplica a cualquier sentencia posible y se genera el correspondiente mutante. De esta forma, a partir de un código de algunas decenas de líneas es posible generar cientos de mutantes. Una vez que se han generado los mutantes, se ejecuta cada caso de prueba sobre el código original y sobre cada uno de los mutantes. Si las salidas obtenidas al ejecutar un caso de prueba t sobre el código original y el mutante m difieren, decimos que el caso de prueba t mata al mutante m. Esta definición puede extenderse al conjunto T de casos de prueba: si algún caso de prueba $t \in T$ mata al mutante m, decimos que el conjunto T mata a m. Un conjunto de casos de prueba bien diseñado debería distinguir entre los mutantes y el programa original y, por tanto, debería ser capaz de matar a muchos de los mutantes. Cuantos más mutantes mate el conjunto de casos de prueba mejor se asume que es la calidad del mismo. Sin embargo, es posible que algunos mutantes queden vivos tras aplicar cualquier conjunto de casos de prueba porque pueden ser funcionalmente equivalentes al código original. Esta equivalencia funcional impide la completa automatización de la medida de calidad del conjunto de casos de prueba. En efecto, comprobar si dos programas son funcionalmente equivalentes o no, es un problema indecidible. En la práctica, lo que se hace es una inspección manual del mutante que se sospecha que es funcionalmente equivalente. Si se puede afirmar que es funcionalmente equivalente se marca como tal y no se considera para el cálculo de la calidad del conjunto de casos de prueba. Empíricamente se ha observado que entre el 5% y el 20% de los mutantes generados son funcionalmente equivalentes al programa original.

Si llamamos M al conjunto de mutantes generados, K a los mutantes que ha matado el conjunto de casos de prueba T y E al conjunto de mutantes funcionalmente equivalentes a P (programa original), definimos el "mutation score" (MS) como el cociente:

$$MS(P,T) = \frac{|K|}{|M| - |E|}$$

El MS es un número entre 0 y 1 que representa la calidad del conjunto de casos de prueba T. Cuanto más alto sea, mayor será la calidad de T. Offutt [22] propuso una estrategia para diseñar casos de prueba basada en los mutantes. Una vez diseñado el primer conjunto de casos de prueba, se ejecuta sobre el programa original y sobre los mutantes. Aquellos mutantes que sobrevivan son inspeccionados para averiguar si son funcionalmente equivalentes al código original, en cuyo caso son descartados. Si no es funcionalmente equivalente se diseña un caso de prueba que sea capaz de distinguir a dicho mutante del código original y luego se añade al conjunto de casos de prueba. Tras haber analizado cada uno de los mutantes supervivientes, se vuelve a ejecutar el conjunto de casos de prueba sobre ellos y se repite todo el proceso. En cualquier momento podría detectarse un error en el código original, en cuyo caso habría que volver a generar todos los mutantes tras modificar el código, y repetir nuevamente el proceso.

B. Desarrollo Dirigido por Modelos

En esta sección se introducen los conceptos fundamentales relacionados con el desarrollo de software dirigido por modelos que se encuentran dentro del área Model Based Engineering (MBE). Se presentan las definiciones de MDE, MDD y MDA, focalizando en la propuesta MDA del OMG. Luego se hace una breve introducción a QVT y sus variantes.

1) Ingeniería Dirigida por Modelos (MDE): Es un paradigma dentro de la ingeniería del software que apoya el uso de modelos y transformaciones entre ellos como piezas clave para dirigir todas las actividades relacionadas con la ingeniería del software. MDE es un término más amplio que MDD [1], ya que la ingeniería dirigida por modelos abarca todas las actividades de la ingeniería de software, y no solo el desarrollo de sistemas. De forma sencilla, un modelo es una abstracción simplificada de un sistema o concepto del mundo real.

2) Desarrollo Dirigido por Modelos (MDD): El desarrollo de software dirigido por modelos es un paradigma de construcción de software cuyas motivaciones principales son la independencia de los desarrolladores de software a través de estandarizaciones y la portabilidad de los sistemas de software. El propósito de MDD es separar el diseño del sistema de la arquitectura de las tecnologías, para que puedan ser modificados independientemente. Para lograr esto, se asigna a los modelos un rol central y activo bajo el cual se derivan modelos, que van desde los más abstractos a los más concretos. El proceso de desarrollo se realiza a través de transformaciones sucesivas e iteraciones. Las transformaciones toman un modelo de entrada (o más de uno) y producen un modelo como salida. Las transformaciones son esenciales en el desarrollo dirigido por modelos. La característica principal de este paradigma radica en que todo debe girar sobre la base de modelos, definidos a partir de metamodelos. El principal objetivo que cumple MDD es tratar de minimizar los costos y el tiempo de desarrollo de las aplicaciones de software, buscando mejorar la calidad, la independiencia de la plataforma donde el software se ejecuta y garantizando las inversiones en tecnología.

3) Arquitectura Dirigida por Modelos (MDA): Es la propuesta concreta del OMG para implementar MDD, usando notaciones, mecanismos, herramientas y estándares definidos por dicha organización. MDA fue la primera de las propuestas de MDD y la que consiguió hacer despegar el uso y adopción de los modelos como piezas clave del desarrollo de software. MDA, al igual que MDD, aboga por la separación de la especificación de la funcionalidad de un sistema independientemente de su implementación en cualquier plataforma tecnológica concreta. Asimismo,se caracteriza por el uso de transformaciones de modelos para convertir unos modelos en otros, hasta obtener las implementaciones finales. Los estándares que el OMG ofrece para realizar MDA incluyen, entre otros:

- UML [3] (Unified Modeling Language) como lenguaje de modelado.
- MOF [1] (Meta-Object Facility) como lenguaje de metamodelado.
- OCL [23] (Object Constraint Language) como lenguaje de restricciones y consulta de modelos.
- QVT [24] (Query-View-Transformation) como lenguaje de transformación de modelos.
- XMI (XML metadata interchange) como lenguaje de intercambio de información.

La noción de metamodelo es fundamental en MDA. Un metamodelo es un modelo de un lenguaje de modelado. Un modelo es una representación de un sistema descripto en algún lenguaje bien definido. Una vista particular (o aspecto) de un sistema puede ser capturada por un modelo, escrito en el lenguaje de su metamodelo. La idea básica detrás de MDA consiste en elaborar primero modelos de muy alto nivel, denominados modelos independientes de las plataformas o PIM (platform-independent models), completamente independientes de las aplicaciones informáticas que los implementarán, o de las tecnologías usadas para desarrollarlos o implementarlos. MDA define entonces algunos procesos para ir refinando esos modelos, particularizándolos progresivamente en modelos específicos de la plataforma a usar; cada vez más concretos conforme se van determinando los detalles finales.

Una transformación de modelos es el proceso de convertir un modelo de un sistema en otro modelo. Una transformación establece un conjunto de reglas que describen como un modelo expresado en un lenguaje origen puede ser transformado en un modelo en un lenguaje destino. La Figura 3 describe los elementos incluidos en una transformación de modelos: un modelo destino B es obtenido desde un modelo origen A, utilizando un motor que ejecuta la especificación de una transformación. El modelo origen A conforma al metamodelo MA; lo mismo ocurre con el modelo destino B respecto a su metamodelo MB. Similarmente, una especificación de una transformación debe conformar a un metamodelo. En resumen, una transformación requiere de : (1) modelos origen y destino, (2) metamodelos origen y destino y, (3) la especificación de la transformación.

C. Herramientas de transformación de modelos

Se describen a continuación distintas herramientas que colaboran con las distintas etapas del proceso de transformación de modelos.

MoDisco [25] es un proyecto de Eclipse GMT (Generative Model Technologies) para la ingeniería inversa dirigida por modelos. El objetivo es permitir extracciones prácticas de modelos desde sistemas heredados. Debido a la muy diferente naturaleza y la heterogeneidad tecnológica de los sistemas heredados, hay varias maneras diferentes para extraer modelos de este tipo de sistemas. MoDisco propone una genérica y extensible aproximación dirigida por metamodelos para obtener modelos. Un framework básico y un conjunto de instrucciones son provistos a los colaboradores de Eclipse para desarrollar sus propias soluciones para obtener modelos. MoDisco es un proyecto colaborativo que involucra a varias organizaciones.

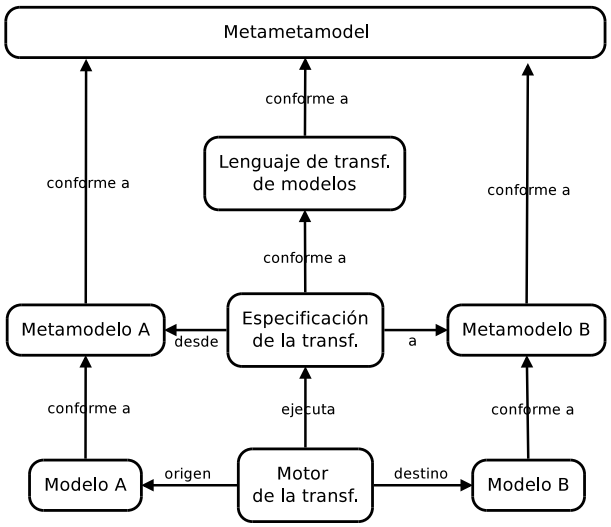

Fig. 2: Transformación de modelos

Cada una de ellas aporta su propia experiencia en un área determinada.

QVT [24] es un conjunto de lenguajes para transformación de modelos, definido por el OMG. La transformación de modelos es una técnica clave usada en la arquitectura dirigida por modelos. En la Figura 3 se visualizan los tres lenguajes de transformación de modelos estándares que define QVT [26]: QVT-Relations, Operational Mapping (QVT Operational) y Black Box. Todos operan sobre modelos que conforman los metamodelos Meta-Object Facility (MOF) 2.0. Una transformación en cualquiera de los tres lenguajes QVT puede ser considerada un modelo, conforme a uno de los metamodelos especificados en el estandar QVT. El estándar QVT integra el estándar OCL 2.0 y lo extiende con características imperativas.

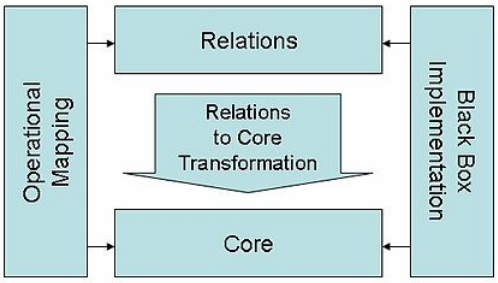

Fig. 3: Arquitectura QVT

QVT-Operational (QVT-O) es la variante imperativa de QVT para implementar una transformación y es, precisamente, la utilizada en este trabajo. La sintaxis es similar a la de otros lenguajes imperativos como JAVA y, a diferencia de la variante declarativa (QVT-Relations), los mapeos operacionales son unidireccionales, es decir, hay un modelo origen y uno o más destinos.

Acceleo [27] es un generador de código open source de la Eclipse Software Fundation que permite usar un enfoque dirigido por modelos para construir aplicaciones. Es una implementación del estandar MOFM2T para realizar transformaciones de modelo a texto. Acceleo provee herramientas para generación de código a partir de modelos basados en EMF. Gracias a estas herramientas permite, por ejemplo, generación incremental. La generación incremental da la posibilidad de generar una parte del código y luego modificarlo para finalmente regenerarlo una vez más sin perder las modificaciones previas. Acceleo también permite:

- Generación de código desde cualquier clase de metamodelo compatible con EMF.
- Personalización de la generación de código con plantillas definidas por el usuario.
- Generación de cualquier lenguaje textual, como C o JAVA, e incluso un reporte técnico.

IV. Implementación de la Transformación con QVT-Operational

En esta sección se presentan las partes relevantes de la transformación que implementa los operadores de mutación. Estas transformaciones describen relaciones entre el metamodelo JAVA (origen y destino en este caso) especificado con el estándar MOF [28].

En el actual trabajo se utiliza QVT-O (implementación imperativa de QVT). Su sintaxis es similar a la de otros lenguajes imperativos. A diferencia de las variantes declarativas, QVT-O es unidireccional, debe haber un único modelo origen y uno o más modelos destino. Tener en cuenta que previamente se obtiene un modelo a partir de un programa JAVA que luego es pasado como parámetro a la transformación.

La transformación consta de un conjunto de "mappings" de las distintas estructuras del metamodelo JAVA. Estos "mappings" especifican cómo un objeto del modelo origen es transformado a un objeto del modelo destino. En este caso, los modelos origen y destino conforman al metamodelo JAVA.ecore, el cual es un reflejo del lenguaje JAVA según la versión 7 [29] de la especificación del lenguaje. Una ejecución de la transformación QVT-O toma como entrada un modelo conforme al metamodelo JAVA y retorna uno o más modelos idénticos al original a excepción de un operador que se modifica por mutación. En este trabajo se implementan operadores de mutación AORU, COD, COI y ROR. A continuación se muestra la parte más importante del código que implementa el proceso de transformación.

Definición del tipo de los modelos utilizados como entrada y salida de la transformación:

```
modeltype JAVA uses 'http://www.eclipse.org/
    MoDisco/Java/0.2.incubation/java';
```

Encabezado de la transformación:

```
transformation transform(in java:JAVA, out
    JAVA);
```

La implementación permite especificar qué operador de mutatión aplicar y sobre cuál de todas sus ocurencias. Esto permite en el futuro automatizar la generación de todos los posibles mutantes para cada operador.

La siguiente declaración representa el orden de aparición en el código del operador a modificar:

```
configuration property levelToModify:Integer;
```

La propiedad `property opToModify` especifica el operador de mutación a aplicar. En el trabajo se implementaron trece operadores del tipo AORU, COR, COI y ROR. Cada uno de ellos tiene asignado un identificador.

```
configuration property opToModify:Integer;
```

Las siguientes propiedades se utilizan para contar las apariciones de cada operador a lo largo del código fuente

```
property plus : Integer = 0;
property minus : Integer = 0;
property times : Integer = 0;
property divide : Integer = 0;
property condAnd : Integer = 0;
property condOr : Integer = 0;
property greater : Integer = 0;
property less : Integer = 0;
property greaterEquals : Integer = 0;
property lessEquals : Integer = 0;
property reminder : Integer = 0;
property notEquals : Integer = 0;
property equals : Integer = 0;
```

El siguiente mapeo define precisamente la aplicación del operador de mutación a aplicar según su orden de ocurrencia, especificado en las propiedades definidas anteriormente:

```
mapping java::InfixExpression::ie2ie() : java
    ::InfixExpression {
  switch {
    //Cambio del operador AND por el OR
    case (self.operator = java::
        InfixExpressionKind::CONDITIONAL\_AND)
        {
      log ('CONDITIONAL\_AND');
      condAnd := condAnd +1;
      if (condAnd = levelToModify and
          opToModify = 5) then {
        operator := java::InfixExpressionKind
            ::CONDITIONAL\_OR
      }else{
        operator := self.operator
      }endif;
    }
    //Cambio del operador OR por el AND
    case (self.operator = java::
        InfixExpressionKind::CONDITIONAL\_OR) {
      log ('CONDITIONAL\_OR');
      condOr := condOr +1;
      if (condOr = levelToModify and
          opToModify = 6) then {
```

```
    operator := java::InfixExpressionKind
        ::CONDITIONAL\_AND
  }else{
    operator := self.operator
  }endif;
}
//Cambio del operador + por el -
case (self.operator = java::
    InfixExpressionKind::PLUS) {
  log ('PLUS');
  plus := plus +1;
  if (plus = levelToModify and opToModify
      = 1) then {
    operator := java::InfixExpressionKind
        ::MINUS
  }else{
    operator := self.operator
  }endif;
}
//Cambio del operador - por el +
case (self.operator = java::
    InfixExpressionKind::MINUS)     {
  log ('MINUS');
  minus := minus +1;
  if (minus = levelToModify and opToModify
      = 2) then {
    operator := java::InfixExpressionKind
        ::PLUS
  }else{
    operator := self.operator
  }endif;
}
//Cambio del operador > por el <
case (self.operator = java::
    InfixExpressionKind::GREATER){
  log ('GREATER');
  greater := greater +1;
  if (greater = levelToModify and
      opToModify = 7) then {
    operator := operator := java::
        InfixExpressionKind::LESS
  }else{
    operator := self.operator
  }endif;
}
//Cambio del operador <= por el <
case (self.operator = java::
    InfixExpressionKind::LESS\_EQUALS) {
  log ('LESS\_EQUALS');
  lessEquals := lessEquals +1;
  if (lessEquals = levelToModify and
      opToModify = 9) then {
    operator := operator := java::
        InfixExpressionKind::LESS
  }else{
    operator := self.operator
  }endif;
}
//Cambio del operador == por el !=
case (self.operator = java::
    InfixExpressionKind::EQUALS) {
  log ('EQUALS');
  equals := equals +1;
  if (equals = levelToModify and
      opToModify = 13) then {
    operator := operator := java::
        InfixExpressionKind::NOT\_EQUALS
  }else{
    operator := self.operator
  }endif;
}
//Cambio del operador \% por el /
case (self.operator = java::
    InfixExpressionKind::REMAINDER) {
  log ('REMAINDER');
  reminder := reminder +1;
  if (reminder = levelToModify and
      opToModify = 11) then {
    operator := operator := java::
        InfixExpressionKind::DIVIDE
  }else{
    operator := self.operator
  }endif;
}
...
...
...
else{
  log ('NOTHING');
  operator := self.operator;
}
};
comments := self.comments -> com2com();
originalCompilationUnit := self.
    originalCompilationUnit.map cu2cu();
originalClassFile := self.originalClassFile.
    map cf2cf();
rightOperand := self.rightOperand.map ex2ex
    ();
leftOperand := self.leftOperand.map ex2ex();
extendedOperands := self.extendedOperands ->
    ex2ex();
}
```

V. Ejemplo de Aplicación

En esta sección se describe un ejemplo del operador de mutación ROR sobre una versión del algoritmo de ordenamiento *BubbleSort*. Particularmente, se reemplaza la ocurrencia del operador $<=$ por el operador $<$ en la condición del ciclo principal. Por simplicidad, se considera aquí la aplicación de un único operador de mutación.

En la primer etapa se utiliza la herramienta MoDisco para obtener el modelo JAVA del algoritmo conforme al metamodelo JAVA en formato Ecore [30]. Luego se aplica la transformación que modifica el modelo JAVA para obtener un modelo JAVA mutado. Posteriormente, se genera el código fuente del modelo JAVA mutado y finalmente se ejecutan los casos de prueba correpondientes para evaluar su nivel de eficacia.

1) Implementación de *BubbleSort.java*.

```
public class BubbleSort {

  public static void BubbleSort (int[]
      num)
  {
    int j, i=1;
    boolean flag = true;
    int temp;
```

```
    while (i <= num.length - 1 && flag)
    {
      flag = false;
      for (j = 0;j < num.length-i ;j++)
      {
        if (num[j] > num[j + 1 ] )
        {
          temp = num[j];
          num[j] = num[j+1];
          num[j+1] = temp;
          flag = true;
        }
      }
    }
  }
}
```

2) **Transformación de texto a modelo mediante MoDisco/Discoverer**
Mediante la herramienta MoDisco [25], la cual toma como parámetro el archivo *BubbleSort.java*, se genera el modelo JAVA **BubbleSortSource.xmi**. La figura 4 ilustra parcialmente el modelo JAVA obtenido en formato Ecore.

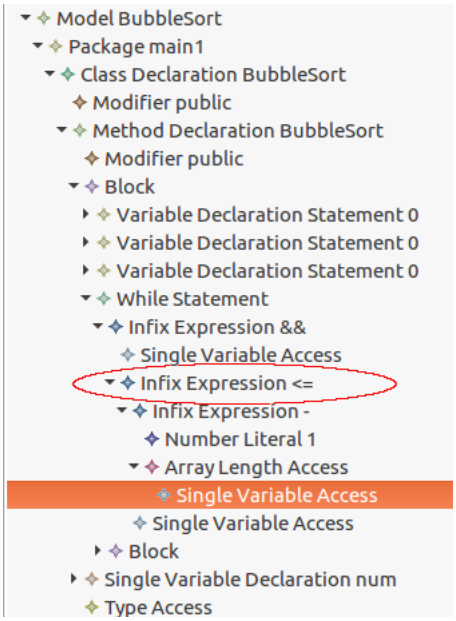

Fig. 4: Modelo JAVA de BubbleSort.

3) **Transformación M2M mediante QVT-O**
Esta etapa del proceso es la más relevante y la principal contribución del trabajo, ya que es la que define la transformación M2M.
La transformación se ejecuta configurando las propiedades levelToModify = 1 y opToModify = 9, y genera un nuevo archivo que representa el modelo mutado **BubbleSort_Mut_"op"_"i".xmi**, donde "op" se reemplaza por el identificador del operador de mutación e "i" es el número de ocurrencia ("9" y "1" para el ejemplo).
La figura 5 ilustra la sección del modelo JAVA mutado donde fue aplicado el reemplazo del operador $<=$ por el operador $<$.

Fig. 5: Modelo JAVA mutado de BubbleSort.

4) **Transformación de Modelo a Texto (M2T) mediante Acceleo**
En esta etapa el modelo obtenido de la transformación M2M es traducido a código fuente JAVA mediante Acceleo [27]. A continuación se ilustran parcialmente las partes relevantes del código que genera la mutación del ejemplo.

```
...
[if (statement.oclIsTypeOf(
    WhileStatement))]
[genWhileStat(statement.oclAsType(
    WhileStatement))/][/if]
...

[template public genWhileStat(aWhile :
    WhileStatement)]
while ([genExpression(aWhile.expression
    )/])
[genStatement(aWhile._body)/]
[/template]
...

[template public genExpression(
    expression : Expression)]
   [if (expression.oclIsTypeOf(
       InfixExpression))]
   [genInfixExpression(expression.
       oclAsType(InfixExpression))/][/if
       ]

...
[template public genInfixExpression(
    infixExp : InfixExpression)]
[genExpression(infixExp.leftOperand)/]
    [infixExp.operator/] [genExpression(
    infixExp.rightOperand)/] [
    genExtOperands(infixExp)/]
[/template]
```

```
...
```

Luego de la transformación M2T se obtiene el archivo JAVA correspondiente *BubbleSort_Mut_9_1.java*.

```
public class BubbleSort_Mut_9_1 {

  public static void BubbleSort_Mut_9_1
       (int[] num)
  {
    int j, i=1;
    boolean flag = true;
    int temp;

    while (i < num.length - 1 && flag)
        // Reemplazo de <= por <
    {
      flag = false;
      for (j = 0;j < num.length-i ;j++)
      {
        if (num[j] > num[j + 1 ] )
        {
          temp = num[j];
          num[j] = num[j+1];
          num[j+1] = temp;
          flag = true;
        }
      }
    }
  }
}
```

5) **Ejecución de casos de prueba mediante JUnit**
El objetivo principal de las pruebas es el hallazgo de errores en el programa bajo prueba, de tal manera que si un conjunto de casos de prueba no encuentra errores no es, muy probablemente, porque el sistema no los tenga, sino porque los casos de prueba están mal diseñados.
Se ejecuta mediante JUnit un pequeño conjunto inicial de casos de prueba a fin de matar al mutante. Dicho conjunto inicial puede construirse bajo algún criterio tradicional de testeo, por ejemplo, cobertura de caminos. Esta etapa no es parte de la contribución principal del trabajo; sin embargo, se describe para completar el ejemplo de aplicación.
Se considera al inicio el siguiente conjunto de casos de test y la implementación parcial de la ejecución de las pruebas:
$T_1 = \{ \{3,2,1,5,4\}, \{3,4,2,1,5\}, \{4,5,3,1,2\}, \{1,2,4,3,5\} \}$

```
@Test
public void testBubbleSort1() {
        //Casos de Prueba
        int[] a = {3,2,1,5,4};
        int[] b = {3,4,2,1,5};
        int[] c = {4,5,3,1,2};
        int[] d = {1,2,4,3,5};

        // Lista para almacenar las
            permutaciones de los
            elementos de un caso.
        LinkedList<int[]> list = new
            LinkedList<int[]>();
        ...
        Perm(a, "", a.length, a.length,
             list);
        BubbleSort.BubbleSort(a);
        Assert.assertTrue(contains(list
            , a) && isOrdered(a));

        Perm(b, "", b.length, b.length,
             list);
        BubbleSort.BubbleSort(b);
        Assert.assertTrue(contains(list
            , b) && isOrdered(b));
        ...
}
```

La ejecución de JUnit con las pruebas de T_1 revelan que estas satisfacen la salida esperada, pero si las ejecutamos sobre la versión mutada del Bubblesort (BubbleSort_Mut_9_1) no son capaces de detectar el mutante, es decir, el mutante permanece vivo.
Atento al cambio inyectado en la mutación, un nuevo diseño del conjunto de casos de prueba es necesario y, en consecuencia, entradas en donde el último elemento de la secuencia es el menor de todos son necesarias para matar al mutante:$\{\{4,5,3,2,1\}, \{3,2,4,5,1\} ...\}$.

VI. Conclusiones y Trabajos Futuros

En este trabajo se propone un mecanismo de aplicación de la técnica *Mutation Testing* en el contexto MDD, vía transformación de modelos de tipos T2M, M2M y M2T, a fin de evaluar un conjunto de casos de prueba. El proceso de transformaciones inicia con un programa escrito en el lenguaje JAVA, el cual es transformado a una representación en formato XMI, precisamente ECORE, conforme al metamodelo de dicho lenguaje de programación. Este proceso es llevado a cabo utilizando la herramienta MoDisco. Sobre dicho modelo se aplican las operaciones convencionales de mutación a través de una transformación escrita en *QVT Operacional*. Finalmente se obtiene el programa JAVA mutado vía una transformación M2T definida con MOF2Text. Los casos de prueba que *maten al mutante* mediante ejecuciones del programa mutado serán considerados buenos casos de prueba. En la literatura del tema existe una gran variedad de indicadores para determinar la efectividad de un caso de prueba. El objetivo de MT es evaluar cuán efectivo es un conjunto de casos de prueba y, de ser necesario, reescribir aquellos que presenten falencias.

Aplicar esta técnica mediante transformación de modelos se diferencia de otras formas convencionales, las cuales, por un lado, se basan en generar árboles de sintaxis abstracta mediante algún parser de JAVA para realizar la mutación. Esto incorpora cierta complicación para la comunidad de ingeniería de software. Por otro lado, el hecho de construir un modelo del programa fuente y trabajar sobre éste evita el costo de realizar múltiples compilaciones de mutantes durante su generación. Además, el modelo Java obtenido del programa original incluye información (metadata) adicional procesada por MoDisco que puede ser de gran utilidad para la generación de mutantes más complejos; por ejemplo, información sobre cantidad de usos para cada definición de variable.

Como trabajo futuro identificamos los siguientes:

- Extender la lista de mutaciones. Por ejemplo, forzar divisiones por cero y otros cambios en el código que afecten su semántica. Considerar la implementación de mutantes más complejos como renombre de variables, haciendo uso de la información adicional (metadata) que se dispone sobre los modelos obtenidos.
- Considerar técnicas de optimización para la generación de mutantes a fin de reducir los costos.
- Automatizar en una única herramienta el proceso completo de transformaciones y testeo.
- Considerar la posibilidad de realizar el mismo proceso sobre otros lenguajes de programación.

REFERENCIAS

[1] S. J. Mellor, A. N. Clark, y T. Futagami, "Guest editors' introduction: Model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 14–18, 2003.

[2] J. Miller y J. Mukerji, "Mda guide version 1.0.1," Object Management Group (OMG), Tech. Rep., 2003.

[3] J. Rumbaugh, I. Jacobson, y G. Booch, *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[4] R. A. DeMillo, R. J. Lipton, y F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978. [Online]. Disponible en: http://dx.doi.org/10.1109/C-M.1978.218136

[5] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, y W. E. Wong, "Model-based mutation testing-approach and case studies," *Sci. Comput. Program.*, vol. 120, no. C, pp. 25–48, May 2016. [Online]. Disponible en: https://doi.org/10.1016/j.scico.2016.01.003

[6] Y.-S. Ma, J. Offutt, y Y. R. Kwon, "Mujava: An automated class mutation system: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, Jun. 2005. [Online]. Disponible en: http://dx.doi.org/10.1002/stvr.v15:2

[7] S. E. G. Brida, "Una herramienta flexible para la mutación de programas java con aplicaciones en la reparación de programas," Universidad Nacional de Río Cuarto, Tech. Rep., 2014.

[8] *The Major mutation framework*, Ultimo Acceso: Marzo de 2017. [Online]. Disponible en: http://mutation-testing.org/

[9] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for Java," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, July 23–25 2014, pp. 433–436.

[10] R. Just, F. Schweiggert, y G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, November 9–11 2011, pp. 612–615.

[11] H. Coles, T. Laurent, C. Henard, M. Papadakis, y A. Ventresque, "Pit: A practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 449–452. [Online]. Disponible en: http://doi.acm.org/10.1145/2931037.2948707

[12] I. Moore, "Jester - a junit test tester." 2001.

[13] L. Madeyski y N. Radyk, "Judy - a mutation testing tool for java," *IET Software*, vol. 4, no. 1, pp. 32–42, 2010. [Online]. Disponible en: http://dx.doi.org/10.1049/iet-sen.2008.0038

[14] D. Schuler y A. Zeller, "Javalanche: efficient mutation testing for java," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 297–298.

[15] Y. Jia y M. Harman, "Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language," TAIC PART, 2008.

[16] P. R. Mateo y M. P. Usaola, "Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, 2012, pp. 646–649. [Online]. Disponible en: http://dx.doi.org/10.1109/ICSM.2012.6405344

[17] M. J. Suarez-Cabal, C. de la Riva, y J. Tuya, "Sqlmutation: A tool to generate mutants of sql database queries," *Mutation Analysis, Workshop on*, vol. 00, p. 1, 2006.

[18] J.-M. Mottu, B. Baudry, y Y. Le Traon, "Mutation analysis testing for model transformations," in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2006, pp. 376–390.

[19] G. M. K. Selim, J. R. Cordy, y J. Dingel, "Model transformation testing: The state of the art," in *Proceedings of the First Workshop on the Analysis of Model Transformations*, ser. AMT '12. New York, NY, USA: ACM, 2012, pp. 21–26. [Online]. Disponible en: http://doi.acm.org/10.1145/2432497.2432502

[20] A. Gonzalez, M. Uva, y M. Frutos, "Refactoring java code by transformation rules in qvt-relation," in *2013 XXXIX Latin American Computing Conference (CLEI), Caracas (Naiguata), Venezuela, October 7-11, 2013*, 2013, pp. 1–9. [Online]. Disponible en: http://dx.doi.org/10.1109/CLEI.2013.6670658

[21] E. Omar, S. Ghosh, y D. Whitley, "Homaj: A tool for higher order mutation testing in aspectj and java," in *Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION'14)*, Cleveland, USA, 31 March 2014.

[22] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 1, pp. 5–20, Jan. 1992. [Online]. Disponible en: http://doi.acm.org/10.1145/125489.125473

[23] J. Warmer y A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[24] OMG, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*, Object Management Group , Rev. 1.1, 2011. [Online]. Disponible en: http://www.omg.org/spec/QVT/1.1/

[25] H. Bruneliere, J. Cabot, F. Jouault, y F. Madiot, "Modisco: A generic and extensible framework for model driven reverse engineering," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 173–174. [Online]. Disponible en: http://doi.acm.org/10.1145/1858996.1859032

[26] P. Guduric, A. Puder, y R. Todtenhoefer, "A Comparison between Relational and Operational QVT Mappings," in *2009 Sixth International Conference on Information Technology: New Generations*, Las Vegas, NV, USA, 2009, pp. 266–271. [Online]. Disponible en: http://0-ieeexplore.ieee.org.innopac.up.ac.za/xpl/downloadCitations

[27] Eclipse. (2012) Acceleo. [Online]. Disponible en: http://www.eclipse.org/acceleo/

[28] *OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1*, Object Management Group , Rev. 2.4.1, 2011. [Online]. Disponible en: http://www.omg.org/spec/MOF/2.4.1

[29] *The Java Language Specification*, Oracle, Last access: abril 2018. [Online]. Disponible en: https://docs.oracle.com/javase/specs/jls/se7/html/index.html

[30] *Ecore metamodel specification*, Object Management Group, Last access: May 2015. [Online]. Disponible en: http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html