UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

GRADO EN INGENIERÍA DEL SOFTWARE

Interoperabilidad de modelos UML entre herramientas gráficas y textuales

UML models interoperability between graphical and textual modeling tools

Realizado por
Julia Robles Medina

Tutorizado por
Antonio Jesús Vallecillo Moreno
Paula Muñoz Ariza

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2021

Resumen

A la hora de diseñar modelos software se puede optar por utilizar herramientas de notación gráfica o de notación textual. La preferencia entre ellas es uno de los grandes debates dentro de la Ingeniería del Software Dirigida por Modelos, en la que se enmarca este proyecto, pero, a pesar de que ambas presentan sus claros beneficios, la interoperabilidad existente entre ellas es casi inexistente. El objetivo de este proyecto es la creación de una herramienta que cambie eso. Esta se centrará, en primera instancia, en las herramientas MagicDraw y USE, aunque utilizando un formato pivote que permita su ampliación a otros lenguajes.

Para su desarrollo se ha aplicado una metodología de trabajo iterativa incremental con la que, finalmente, se ha logrado una herramienta que permite transformar desde clases simples hasta máquinas de estados, utilizando como formato pivote Eclipse UML2 XMI, que es la adaptación al plugin de Eclipse del estándar XMI. Asimismo, el programa es capaz de corregir errores de equivalencia, normalmente provocados por la incompletud del modelo por parte del usuario.

Todo esto se ha implementado usando Xtext, donde se ha descrito la gramática de USE, y los plugins de Eclipse UML2 y EMF, que han permitido obtener la información procedente de los archivos pivote. Además, se ha desarrollado un algoritmo de equivalencia de modelos, específico para los requisitos de esta aplicación, que permite la automatización de pruebas unitarias.

Palabras clave: USE, MagicDraw, Ingeniería de Software Dirigida por Modelos, Intercambio de modelos, Xtext

Abstract

When designing software models, you can choose to use graphic notation or textual notation tools. The preference between them is one of the great discussions within the Model-driven Engineering, in which this project is framed, but, despite the fact that both present their clear benefits, the existing interoperability between them is almost non-existent. The goal of this project is to create a tool that changes that. This will focus, in first instance, on the MagicDraw and USE tools, although using a pivot format that will allow its extension to other languages.

For its development, an incremental iterative methodology has been applied, finally obtaining a tool that allows transforming from simple classes up to state machines, using Eclipse UML2 XMI as a pivot format, which is the adaptation to the Eclipse plugin of the XMI standard. Likewise, the program is capable of correcting equivalence errors, normally caused by the incompleteness of the model by the user.

All this has been implemented using Xtext, where the USE grammar has been described, and the Eclipse UML2 and EMF plugins, which have made it possible to obtain the information from the pivot files. In addition, a model equivalency algorithm has been developed, specific to the requirements of this application, which allows the automation of unit tests.

Keywords: USE, MagicDraw, Model-driven Engineering, Model exchange, Xtext

Índice

1. Introducción 1
   1.1. Objetivos 1
   1.2. Motivación 2
   1.3. Entregables a desarrollar 3
   1.4. Metodología y fases de trabajo 4
      1.4.1. Metodología 4
      1.4.2. Fases de trabajo 6
   1.5. Estructura de la memoria 7
2. Estado del arte 11
   2.1. Ingeniería del Software Dirigida por Modelos 11
   2.2. Lenguajes Específicos de Dominio 13
   2.3. Intercambio de modelos 14
3. Tecnologías utilizadas 15
   3.1. Java y Eclipse 15
   3.2. Plugin Eclipse UML2 16
   3.3. Plugin Eclipse OCL 17
   3.4. Xtext 17
   3.5. Xtend 18
   3.6. MagicDraw 18
   3.7. USE: UML-based Specification Environment 19
   3.8. JUnit 20
4. Análisis de lenguajes 21
   4.1. Comparación entre modelos 21
   4.2. Diferencias remediables 35

4.3. Limitaciones 37
5. Diseño de la herramienta 39
5.1. Diagrama de componentes 39
5.2. Diagrama de clases 40
5.3. Diagramas de secuencia 43
6. Implementación 51
6.1. Conversión de UML a USE 51
6.2. Conversión de USE a UML 53
6.2.1. Transformación del código OCL 54
6.3. Unificación de conversiones e interfaz 56
7. Validación y pruebas 59
7.1. Algoritmo de equivalencia de modelos 59
7.2. Desarrollo de pruebas 62
7.2.1. Plan de pruebas 62
7.2.2. Ficheros generados 63
7.2.3. Pruebas finales 64
8. Conclusiones 67
8.1. Desarrollo de la herramienta 67
8.2. Líneas futuras de ampliación 69
9. Referencias 71
Apéndice A. Manual de instalación 77
Apéndice B. Manual de usuario 79
Apéndice C. Diagrama de clases modelo completo 87
Apéndice D. Diagrama de clases pivote T completo 89
Apéndice E. Requisitos finales 91
Requisitos funcionales 91
Requisitos no funcionales 97
Apéndice F. Plan de pruebas 99

1

Introducción

Durante este capítulo se realizará un primer acercamiento al proyecto, comentando principalmente la finalidad del mismo, los motivos que lo han incentivado y cómo se ha llevado a cabo.

1.1. Objetivos

El objetivo principal de este proyecto ha sido el desarrollo de una herramienta que permitiese al usuario el intercambio en ambos sentidos de modelos software entre herramientas gráficas y textuales de modelado UML [1].

Inicialmente se ha desarrollado para las herramientas de modelado USE [2] y MagicDraw (MD) [3], aunque para favorecer futuras ampliaciones se ha utilizado una representación intermedia que ha hecho la función de pivote entre ambas.

En cuanto a funcionalidad, la herramienta debía ser capaz de leer, analizar e intercambiar todo tipo de modelos, desde los más simples a los más complejos,

teniendo en cuenta las semejanzas y diferencias entre ambas gramáticas y evitando la generación de modelos incorrectos aunque equivalentes. Todo esto se explicará con mayor detenimiento en el capítulo 4 de este documento.

Asimismo, el hecho de ser un Trabajo de Fin de Grado también ha implicado ciertos objetivos pedagógicos, como el aprendizaje y estudio de nuevos conceptos y herramientas relativas a lenguajes y gramáticas, fundamentalmente referidas a modelado UML, y la aplicación de todo lo aprendido durante la carrera.

1.2. Motivación

Desde los inicios de la ingeniería de software dirigida por modelos (ISDM) se ha planteado la pregunta sobre qué notación, si la visual o la textual, es mejor a la hora de representar los modelos de software [4]. Ejemplo de ello es como, a lo largo del tiempo, se ha desarrollado un considerable abanico de herramientas textuales [5] para representar modelos UML, cuya notación por defecto es visual.

La notaciones textuales guardan la ventaja de tener una curva de aprendizaje menor que las visuales, resultar más cómodas para los programadores y ser más eficientes para la realización de bocetos (*sketches*) de los modelos en las etapas iniciales del diseño. Además, algunas pueden llegar a proporcionar un análisis en profundidad del modelo y/o permitir el modelado del comportamiento del sistema mediante la simulación de modelos de objetos, como ocurre en el caso de USE.

Por su parte, las notaciones visuales permiten una comprensión mucho más rápida y sencilla de los modelos, a la vez que simplifican en gran medida las operaciones que se realizan sobre ellos, como el mantenimiento o la refactorización.

La existencia de una herramienta que facilitase la transformación de modelos de una notación a otra permitiría aprovechar las ventajas y las capacidades de

análisis de cada una de ellas. Para una comparación más detallada de estas notaciones se puede consultar el siguiente trabajo [6].

Por otra parte, algunas universidades, como la de Málaga, enseñan a desarrollar modelos UML en ambas notaciones. A día de hoy, esto conlleva que los alumnos tengan que copiar manualmente sus modelos para pasar de una notación a otra, ya que hay una evidente falta de interoperabilidad entre notaciones y herramientas. Hasta el momento solo existían estándares, como XMI, para el intercambio de modelos visuales, pero no se disponía de ninguna forma de transformar modelos visuales a textuales y viceversa. Solucionar esta problemática es justo lo que se pretende conseguir con este proyecto.

1.3. Entregables a desarrollar

En este apartado se definirá el listado de entregables cuya correcta resolución es necesaria para poder dar por finalizado el proyecto.

Entregable 1. Herramienta desarrollada, capaz de intercambiar correctamente modelos entre MagicDraw y USE en ambos sentidos.

Entregable 2. Informe técnico que contenga la arquitectura y el diseño de la aplicación, así como los artefactos software que sean necesarios (código, ficheros de pruebas, etc.)

Entregable 3. Manual de usuario, es decir, documento sin tecnicismos que permita a cualquier usuario comprender fácilmente la herramienta y utilizarla sin mayores problemas.

Entregable 4. Informe detallado que exponga todo el proceso, desde la organización inicial hasta la redacción de los informes finales.

Al primer entregable se puede acceder tanto en los archivos suministrados como en el repositorio: https://github.com/juliarobles/model-converter, mientras que el resto de entregables están contenidos en esta memoria. Concretamente, el tercer entregable constituye el apéndice B.

1.4. Metodología y fases de trabajo

1.4.1. Metodología

Para llevar a cabo este proyecto se ha empleado una metodología de trabajo iterativa incremental, consistente en la división del proceso en iteraciones y la entrega de un prototipo funcional del producto tras cada una de ellas. Esta metódica, perteneciente a las metodologías agiles, permite evaluar de manera constante el progreso del proyecto, favoreciendo su adaptación a los posibles cambios y errores que puedan surgir durante el mismo y mejorando, a su vez, la calidad del producto gracias a la interacción con el cliente.

En el ámbito de este proyecto se ha optado por esta metodología en vista de la inexactitud del mismo en cuestiones de alcance y dificultad. Gracias a ello se han producido versiones intermedias, totalmente funcionales, que han proporcionado cierta seguridad ante posibles complicaciones en las siguientes iteraciones. Estos prototipos han ido ampliando funcionalidad en cada iteración, terminando finalmente con la obtención del producto final.

Previo al comienzo del desarrollo se realizó un diagrama de Gantt que, a pesar de la naturaleza ágil de la metodología, resultó útil como recurso orientativo para favorecer una organización realista respecto al plazo y las horas disponibles.

Como se puede ver en la Figura 1, situada en la página 10, se estimó un desarrollo basado en siete iteraciones de dos semanas cada una, principalmente centradas en

requisitos de alta, media y baja prioridad que realmente serían identificados de manera general en la iteración 0.

A pesar de que el desarrollo no ha sido llevado a la práctica siguiendo al pie de la letra el diagrama, sí que ha terminado siendo bastante coincidente. Finalmente se han realizado siete iteraciones de entre una a tres semanas cuyos objetivos concretos se explicarán con más detalle a continuación.

Iteración 0 – Análisis y prueba de herramientas

Iteración inicial que no genera ningún prototipo. Centrada principalmente en la puesta en marcha del proyecto, la investigación de las posibles soluciones y la realización de un diseño general de la aplicación.

Iteración 1 – Desarrollo de requisitos de alta prioridad de MD a USE

Busca transformar correctamente modelos simples procedentes de MagicDraw a archivos USE legibles por la herramienta. Un modelo se considera simple cuando únicamente contiene clases, relaciones, atributos, operaciones e invariantes sin OCL [7] ni SOIL [8].

Iteración 2 – Desarrollo de requisitos de alta prioridad de USE a MD

Pretende convertir correctamente modelos simples escritos en lenguaje USE, y legibles por la herramienta, a un formato que MagicDraw permita importar sin problemas.

Iteración 3 – Desarrollo de requisitos de media-alta prioridad

Centrada en la integración de OCL en la herramienta por ambos sentidos.

Iteración 4 – Desarrollo de requisitos de media-baja prioridad

Incluye todo lo referente a la interfaz y la interacción de la aplicación con el usuario.

Iteración 5 – Desarrollo de requisitos de baja prioridad

Gira en torno a la conversión en ambos sentidos de máquinas de estados de protocolo que incluyan únicamente estados simples, ni compuestos ni de submáquinas.

Iteración 6 – Pruebas finales y desarrollo de la memoria

Última iteración enfocada en la realización de pruebas y la corrección de errores, con el objetivo de garantizar el correcto funcionamiento del producto final. Incluye, también, la confección final de esta memoria.

Sumado a esto hay que destacar que se concertaron reuniones cada una o dos semanas, siendo dos el plazo máximo, para revisar el avance del proyecto y concretar cómo se debería avanzar.

1.4.2. Fases de trabajo

En cada una de las iteraciones mencionadas con anterioridad se llevaron a cabo, en mayor o menor medida, las siguientes seis fases de trabajo.

Fase 1 - Especificación de requisitos

Análisis de las características de ambas gramáticas con el objeto de obtener una lista de requisitos que describa el comportamiento que debe cumplir la herramienta al final de la iteración.

Fase 2 - Estudio de las herramientas a utilizar

Investigación de las herramientas disponibles y su funcionamiento para su correcto aprovechamiento en el desarrollo del proyecto.

Fase 3 - Diseño de la solución

Diseño y modelado de la aplicación con el propósito de definir de manera clara la solución que se va a llevar a cabo respecto a los requisitos obtenidos en la primera fase y el estudio realizado en la segunda.

Fase 4 – Implementación

Codificación de los requisitos identificados respecto al diseño realizado.

Fase 5 – Pruebas

Verificación y validación de que la versión de la aplicación obtenida cumple con el comportamiento especificado tanto en esta iteración como en las anteriores.

Fase 6 - Escritura de la memoria y documentación

Redacción de la memoria y la documentación con el fin de dejar constancia y detallar lo realizado durante la iteración, ayudando con ello a su futura comprensión o reutilización.

1.5. Estructura de la memoria

Este documento ha sido estructurado siguiendo, esencialmente, las fases enumeradas en el anterior apartado. A continuación se realizará un pequeño resumen de cada uno de los capítulos siguientes.

Capítulo 2 – Estado del arte

Este capítulo encuadrará al lector en el contexto del proyecto, definiendo y explicando las características más relevantes de los entornos relaciones con el mismo, siendo en este caso la Ingeniería del Software Dirigida por Modelos, los Lenguajes Específicos de Dominio y lo que concierne al intercambio de modelos.

Capítulo 3 – Tecnologías utilizadas

Se introducirán todas las tecnologías utilizadas para el desarrollo de la herramienta, comentando para qué se han empleado en cada caso.

Capítulo 4 – Análisis de lenguajes y requisitos finales

Incluirá un análisis detallado de las equivalencias entre ambas gramáticas y el formato pivote, gracias al cual se han obtenido los requisitos utilizados para este desarrollo. Además, se informará de las soluciones tomadas respecto a ciertas diferencias y de las limitaciones de la herramienta.

Capítulo 5 – Diseño de la herramienta

Este capítulo contendrá los modelos realizados durante el desarrollo. Concretamente incluirá un diagrama de componentes con la relación entre las distintas tecnologías utilizadas, un diagrama de clases de la aplicación y varios diagramas de secuencia que muestren la interacción del usuario con la herramienta.

Capítulo 6 – Implementación

Se explicará cómo se ha implementado la herramienta respecto a cada uno de los sentidos, los problemas que hayan podido surgir y la unificación final en una interfaz.

Capítulo 7 – Validación y pruebas

En este capítulo se hablará sobre las pruebas realizadas, argumentando cómo se ha comprobado que dos modelos son equivalentes y justificando el porqué de los ficheros de prueba generados.

Capítulo 8 – Conclusiones

Finalmente se expondrán las conclusiones a las que se ha llegado durante el desarrollo de la herramienta. Además, se comentarán algunas posibles líneas de trabajo futuro que extenderían o mejorarían el funcionamiento actual de la herramienta.

Apéndice A – Manual de instalación

En este documento se explicará cómo se debe instalar la aplicación y los requisitos que son necesarios cumplir para ejecutarla correctamente.

Apéndice B – Manual de usuario

Este apéndice constituye uno de los entregables del desarrollo. En él se describirán los pasos que cualquier usuario de la aplicación debe seguir para utilizarla, junto con una serie de recomendaciones y las limitaciones que posee la herramienta.

Apéndice C – Diagrama de clases modelo completo

Contendrá únicamente un diagrama de clases completo, que incluirá todas las operaciones utilizadas, de la parte correspondiente al modelo de la herramienta.

Apéndice D – Diagrama de clases pivote T completo

Al igual que el anterior apéndice, este contendrá el modelo completo del pivote T desarrollado para las pruebas, con todas sus operaciones.

Apéndice E – Requisitos finales

En él se expondrán todos los requisitos finales de la herramienta, tanto funcionales como no funcionales.

Apéndice F – Plan de pruebas

Este apéndice contiene el listado de casos de pruebas especificados para la herramienta y utilizados para la generación de ficheros de pruebas.

Figura 1. Diagrama de Gantt inicial del desarrollo.

2

Estado del arte

Este capítulo realizará una breve presentación de los ámbitos que atañen a este proyecto: la Ingeniería del Software dirigida por Modelos, los Lenguajes Específicos de Dominio y el intercambio de modelos.

2.1. Ingeniería del Software Dirigida por Modelos

La Ingeniería del Software Dirigida por Modelos (ISDM), o en inglés *Model-driven Engineering* (MDE), se define como un paradigma de la Ingeniería del Software cuya base es la obtención y aplicación de modelos en todas las fases del desarrollo, con la pretensión de sustituir total o parcialmente la programación [9].

Este enfoque surge a raíz de la constante evolución de la tecnología y el auge de los lenguajes de programación, que suponen una enorme inversión de tiempo y costes tanto en lo que a aprendizaje y adaptación se refiere como por la complejidad de codificar, actualizar y mantener el sistema desarrollado. Todo esto

generó la necesidad de abstraer lo máximo posible el sistema de su implementación [10], al igual que se hizo años atrás con la invención de los lenguajes de alto nivel.

Frente a esto, la solución que propone este paradigma es la generación automática del código de un sistema a partir de los modelos diseñados para el mismo, convirtiéndose el modelo en el producto principal a obtener. Lograr esto supondría un gran ahorro, ya que, no solo dejaría de ser prioritaria una de las fases más extensas de la Ingeniería del Software, sino que, además, se podría reutilizar el mecanismo de transformación para otros proyectos, eliminando la necesidad de empezar desde cero. Asimismo, el propio modelo se utilizaría para documentar el sistema, con la ventaja de que siempre permanecería actualizado.

De forma más concreta tenemos la aplicación de esta disciplina por parte de la OMG [11], que recibe el nombre de *Arquitectura Dirigida por Modelos* (MDA, del inglés *Model-driven Architecture*) [12]. Esta, haciendo uso de los modelos UML y otros estándares del consorcio, proporciona directrices para representar las especificaciones del sistema de tal forma que se pueda evolucionar consistentemente hasta obtener el producto final.

Para ello MDA define tres tipos de modelos. Primeramente tenemos los modelos CIM (*Computationally-Independent Model*), que son completamente independientes de la codificación y representan el dominio del problema, es decir, toda la información necesaria previa al desarrollo. En segundo lugar están los modelos PIM (*Platform-Independent Models*), que, tras haber pasado por las fases de análisis y especificación de requisitos, muestran la solución software independientemente de la plataforma. Por último, están los modelos PSM (*Platform-Specific Models*), que se derivan de los anteriores y son específicos para ciertas tecnologías [13]. Las traducciones entre modelos se realizan mediante herramientas específicas y automatizadas, las cuales pueden seguir el lenguaje

estándar de transformación de modelos *Query/View/Transformation* (QVT) [14] propuesto por la OMG.

Otro elemento bastante frecuente en la ISDM son los lenguajes específicos de dominio, de los que se hablará a continuación.

2.2. Lenguajes Específicos de Dominio

Los Lenguajes Específicos de Dominio (DSL, del inglés *Domain-specific Language*), a diferencia de los de propósito general, son lenguajes caracterizados por pertenecer a un dominio concreto y, por consiguiente, admitir un conjunto limitado de tareas [15].

Esto no es ninguna desventaja ya que, al estar centrado en un único dominio y en unas funciones concretas, pueden conseguir que estas se ejecuten de una manera mucho más eficiente y sencilla que utilizando un lenguaje de propósito general. Además, los DSL pueden analizarse mucho mejor, incluso frente a errores, suelen tener una curva de aprendizaje muy baja y, en general, son mucho más seguros [15]. Es por eso por lo que, en determinados escenarios, es importante plantearse la creación de un lenguaje específico de dominio como una posible solución.

Estas situaciones suelen ser desarrollos que impliquen el uso de comandos, la descripción o representación de elementos y/o la definición de reglas [15], aunque realmente se pueden utilizar DSL para un sinfín de objetivos. Además, se debe tener en cuenta la ventaja de que, al utilizar un DSL, se abstrae la lógica de la aplicación, facilitando la comprensión y productividad de los usuarios que lo utilicen y permitiendo modificar la parte técnica sin afectar directamente al lenguaje.

Por ejemplo, el lenguaje que utiliza la herramienta USE para representar los modelos UML es un DSL, al igual que OCL también lo es. Asimismo, los DSL también pueden ser visuales, como es el caso de UML que, a pesar de su envergadura, puede considerarse como tal.

2.3. Intercambio de modelos

Este ámbito es el que tratamos más cercanamente en el proyecto y su concepto consiste, sencillamente, en la capacidad de importar y exportar un modelo entre varias herramientas CASE.

Respecto a este tema no existen demasiadas alternativas. Por ahora, el estándar más utilizado y reconocido es el llamado *XML Metadata Interchange* (XMI), perteneciente a la OMG. Este se caracteriza por representar cualquier elemento que pueda ser expresado en MOF [16] mediante XML [17], aunque usualmente se utiliza para el intercambio de modelos UML. MOF (*MetaObject Facility*), por su parte, es el estándar de OMG encargado de la representación de metadatos y constituye el meta-metamodelo de UML.

El resto de formatos que suelen aceptar las herramientas CASE de definición de modelos, como MagicDraw, se basan en XMI o no están estandarizados, sino que son formatos concretos de exportación de otras herramientas.

Además, este estándar parece haber sido adoptado únicamente por herramientas de notación visual, ya que las textuales, o al menos las más comunes, no incluyen la opción de exportar e importar en dicho formato, quizás por el hecho de la mayoría utilizan sus propios DSLs no basados en MOF.

Tras su respectiva investigación, actualmente no parece que existan estudios o alternativas similares a lo expuesto en este proyecto.

3

Tecnologías utilizadas

3.1. Java y Eclipse

Creado por la empresa Sun Microsystem en 1995, Java es tanto un lenguaje de programación orientado a objetos como una plataforma informática [18].

Se caracteriza fundamentalmente por ofrecer la posibilidad de que un mismo programa o página web pueda ser ejecutado en cualquier dispositivo, gracias a la portabilidad de la Máquina Virtual de Java (JVM) [19]. Esto no siempre es una ventaja, ya que si el dispositivo no tiene Java instalado no funcionará.

Otra de las ventajas de Java es su extensa y activa comunidad, que, unido a la funcionalidad que ya trae por defecto, pone a la disposición del programador una inmensa cantidad de recursos actualizados que facilitarán con creces su labor [19].

Por otro lado tenemos Eclipse, famoso por ser el entorno de desarrollo integrado (IDE) más utilizado para trabajar con Java, aunque también permite otro lenguajes [20]. Destaca primordialmente por la cantidad de complementos (*plugins*) de los que dispone, gracias a su amplia comunidad.

Especialmente por la existencia de ciertos plugins de modelado, que han sido claves en el desarrollo de la herramienta y de los que se hablará a continuación, se han elegido estas dos tecnologías.

Por último, destacar que se ha utilizado una versión de Eclipse que ya trae instaladas todas las herramientas de modelado necesarias [21].

3.2. Plugin Eclipse UML2

El plugin UML2 es una implementación basada en EMF que sigue el metamodelo de UML (Unified Modeling Language), concretamente en su segunda versión [22].

EMF, por su parte, es un marco de trabajo (*framework*) y generador de código capaz de generar el conjunto de clases Java correspondiente a una especificación XMI recibida, permitiendo analizar el modelo fácilmente [23]. Asimismo, también incluye otras clases y funcionalidades para la visualización y edición de modelos, aunque en este proyecto no han sido utilizadas.

A partir de EMF se han desarrollado un gran número de tecnologías y marcos de trabajo, convirtiéndolo en un estándar. Este es el caso tanto del plugin UML2 como de Xtext.

En cuanto al proyecto, MagicDraw permite exportar e importar modelos a formato *Eclipse UML2 XMI*, lo que ha significado que el paso de MagicDraw a USE se base íntegramente en este plugin. Además, este formato se ha convertido en el pivote de la herramienta, ya que es un estándar de intercambio de modelos y otras herramientas visuales, como Papyrus [24], también tienen la opción de exportar e importar en dicho formato.

3.3. Plugin Eclipse OCL

Este plugin también es una implementación basada en EMF, pero esta vez respecto al metamodelo de OCL (Object Constraint Language) [25].

Cabe destacar este plugin ya que, aunque no se ha llegado a utilizar directamente, se ha hecho uso de un archivo de su código fuente, bajo ciertas modificaciones, para complementar la gramática que se ha creado para USE. Esto ha posibilitado la lectura del código OCL que normalmente incluyen los modelos USE.

3.4. Xtext

Xtext se define como "un marco de trabajo para el desarrollo de lenguajes de programación y Lenguajes Específicos de Dominio" [26]. Esta tecnología cuenta con su propio lenguaje para definir gramáticas, a partir de la cual se generan todas las clases necesarias para analizar, editar, validar, generar y compilar archivos escritos en ella.

En definitiva, Xtext permite construir un lenguaje completo y sus herramientas de manera rápida, sencilla y con un buen rendimiento. Además, es totalmente personalizable, ya que las clases generadas se pueden adaptar sin ningún tipo de problema [26].

Concretamente, en este proyecto, se ha utilizado este *framework* para crear la gramática de USE y así poder leer y analizar los archivos recibidos en este formato.

3.5. Xtend

Xtend es una "modernización" de Java, es decir, es un lenguaje de programación basado en Java que añade flexibilidad y expresividad. También añade funciones, como la inferencia de tipos y la sobrecarga de operadores, que facilitan y simplifican la programación. Asimismo, Xtend garantiza el uso de cualquier librería disponible para Java, ya que sus ficheros se compilan en código compatible con Java 8 [27].

Dentro del proyecto este lenguaje se ha utilizado, únicamente, para la generación del fichero Eclipse UML2 XMI correspondiente al archivo USE recibido, tras ser leído y analizado por Xtext. Esto se ha hecho así con la intención de aprovechar el generador de ficheros que crea Xtext automáticamente y que utiliza por defecto código Xtend para determinar a qué y de qué manera se deben convertir los objetos EMF obtenidos.

3.6. MagicDraw

Desarrollada por la compañía No Magic, MagicDraw es una herramienta CASE cuyo principal objetivo es facilitar el análisis y diseño tanto de sistemas como de bases de datos orientadas a objetos. Para ello proporciona una interfaz muy intuitiva y completa que permite la creación visual de una gran variedad de modelos. Por otra parte, otra ventaja de esta herramienta es su compatibilidad con los estándares de UML 2 y XMI [3].

Aunque no han sido utilizadas en este proyecto, es destacable la existencia de otras funcionalidades interesantes como la generación automática de código y la obtención de modelos mediante ingeniería inversa [3].

MagicDraw ha formado parte de este proyecto como la herramienta de notación visual cuyos modelos debían de poder ser transformados correctamente a notación textual y que, a su vez, debía aceptar modelos transformados por la herramienta. Así, MagicDraw se ha utilizado para la exportación de modelos de prueba en formato Eclipse UML2 XMI y la importación de modelos en el mismo formato.

3.7. USE: UML-based Specification Environment

Como su nombre indica, USE es una herramienta para la especificación y validación de sistemas basada en UML y OCL [2]. Para utilizarla es necesario describir el modelo de forma textual en un lenguaje de especificación propio muy sencillo, pudiendo añadir restricciones con OCL y cierta funcionalidad con SOIL.

SOIL, cuyas siglas significan *Simple OCL-based Imperative Language*, es un lenguaje de programación imperativo también particular de USE. Se utiliza para describir un comportamiento paso a paso, uniendo expresiones OCL mediante operadores y funciones propias [28].

Tras especificar el modelo y cargarlo correctamente en la herramienta, USE permite ejecutar las operaciones disponibles junto a comandos SOIL, generando los modelos de secuencia y objetos correspondientes y comprobando en todo momento que las restricciones se cumplen. Todo esto hace de USE una potente herramienta de validación de modelos.

Dentro del proyecto, USE ha sido la herramienta de notación textual elegida para ser transformada a notación visual y viceversa, teniendo el mismo cometido que MagicDraw.

3.8. JUnit

JUnit es el marco de trabajo más popular para automatizar pruebas unitarias en aplicaciones Java. Actualmente van por su quinta versión, que ha sido la utilizada durante el desarrollo [29].

Las pruebas unitarias son aquellas que comprueban el correcto funcionamiento de unidades del sistema, normalmente respecto a distintos datos de entrada. Lo que es una unidad del sistema no está definido de forma fija, sino que se decidirá dependiendo de las necesidades del programa a probar. Estas pruebas son realmente útiles para encontrar errores, aunque, como el resto de métodos, no pueden asegurar la total inexistencia de los mismos [30].

La automatización de estas pruebas supone un enorme ahorro de tiempo lo que, unido a su estupenda integración con Eclipse, ha hecho que esta tecnología sea la elegida para validar el funcionamiento de la herramienta.

4

Análisis de lenguajes

En este apartado se realizará una comparación detallada entre ambas gramáticas y el pivote elegido, concretando cuáles son sus equivalencias y diferencias. Estas últimas se listarán en el segundo y tercer apartado diferenciándolas como corregibles, junto a su respectiva solución, y limitantes. Por último, a raíz de este capítulo se han ido especificando los requisitos del programa, que se pueden consultar en el apéndice E.

4.1. Comparación entre modelos

A continuación se nombrará uno por uno los elementos considerados en el análisis, comentando cómo se expresan en cada lenguaje junto a un pequeño ejemplo. En general, MagicDraw y Eclipse UML2 XMI pueden definir modelos de forma

mucho más minuciosa que USE, por lo que toda esta información faltante no se detallará.

Elemento 1. Modelo.

En USE el modelo se inicializa con la palabra *model* seguida del nombre del modelo. Siempre debe ser la primera palabra del fichero y seguidamente se escribirán el resto de elementos.

```
model NombreModelo
```

En MagicDraw el modelo se inicializa al crear el proyecto y su nombre se ve tanto en la parte superior de los diagramas como en la tabla de contenidos, donde se puede cambiar o consultar el objeto completo.

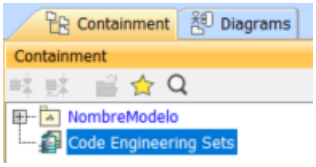


Figura 2. Representación MagicDraw para Modelo

Al exportarlo a XMI, el modelo es un elemento *uml:Model* incluido dentro de una cabecera XMI que inicializa el archivo. Asimismo, dentro del modelo se empaquetan todos sus elementos.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="20131001" xmlns:xmi="http://www.omg.org/spec/XMI/20131001" xmlns:xsi="http://www.w3.org/2001/XMLSchema-i
  <uml:Model xmi:id="eee_1045467100313_135436_1" name="NombreModelo" visibility="public" URI="" viewpoint="">
    ...
  </uml:Model>
  <MagicDrawProfile:auxiliaryResource xmi:id="_YS8CUMlmEeuCJOwSlb8_Pg" base_Element="magicdraw_uml_standard_profile_v_0001"/>
</xmi:XMI>
```

Figura 3. Eclipse UML2 XMI para Modelo

Elemento 2. Enumerado.

En USE los enumerados se especifican con la palabra *enum* seguida del nombre del enumerado, que no puede estar vacío, y de los valores del mismo, conteniendo como mínimo uno.

```
enum Nombre {Value1, Value2, Value3, Value4, Value5}
```

En MagicDraw se representan con un elemento *Enumeration* que también tiene nombre, además de otras características, y puede o no contener valores definidos en componentes de tipo *EnumerationLiteral*.

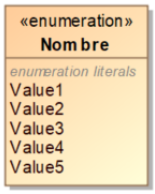


Figura 4. Representación MagicDraw para Enumerado

Al exportarlo a XMI, el enumerado es un elemento empaquetado dentro del modelo, de tipo *uml:Enumeration*. Cuenta con un atributo en su cabecera que incluye su nombre y contiene elementos *ownedLiteral* correspondientes a sus valores, los cuales aparecen como nombre de los mismos.

```
<packagedElement xmi:type="uml:Enumeration" xmi:id="_19_0_2_b7002e9_1623267721498_925415_4894" name="Nombre" visibility="public">
  <ownedLiteral xmi:id="_19_0_2_b7002e9_1623267728907_989785_4916" name="Value1" visibility="public"/>
  <ownedLiteral xmi:id="_19_0_2_b7002e9_1623267734722_585156_4918" name="Value2" visibility="public"/>
  <ownedLiteral xmi:id="_19_0_2_b7002e9_1623267740391_330771_4920" name="Value3" visibility="public"/>
  <ownedLiteral xmi:id="_19_0_2_b7002e9_1623267747126_131184_4922" name="Value4" visibility="public"/>
  <ownedLiteral xmi:id="_19_0_2_b7002e9_1623267752656_163561_4924" name="Value5" visibility="public"/>
</packagedElement>
```

Figura 5. Representación Eclipse UML2 XMI para Enumerado.

Elemento 3. Clase.

En USE las clases se especifican con la palabra *class* seguida del nombre de la clase, que no puede estar vacío. Si la clase es abstracta se añadirá *abstract* antes de *class* y, en el caso de que herede de una o más clases, se añadirá un símbolo *menor que* (<) seguido del nombre de todas las clases separadas por comas.

Los elementos de la clase se incluyen tras su declaración, agrupándose por tipos (atributos, operaciones, etc.) hasta que aparece la palabra *end*. No hay problema en añadir la cabecera de un tipo (ej.: *attributes*) y que esta no contenga nada, aunque, si aparecen, es necesario que sigan el orden que se muestra en el ejemplo.

```
abstract class A < B
        attributes
        operations
        constraints
        statemachines
end
```

En MagicDraw las clases se representan con un elemento *Class* que tiene un nombre y, además de muchas otras características, cuenta con un campo *Is Abstract* para indicar si es abstracta. En su interior contiene sus atributos, operaciones y restricciones. El nombre de la clase se permite que sea vacío. Si la clase hereda de otra u otras se representará con relaciones *Generalization*.

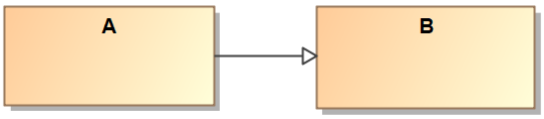


Figura 6. Representación MagicDraw para Clase.

Al exportarlo a XMI, la clase es un elemento empaquetado dentro del modelo, de tipo *uml:Class*. Cuenta con atributos en su cabecera que indican su nombre y si es una clase abstracta (atributo isAbstract), mientras que define todos sus elementos en su interior. Además, contendrá un elemento *generalization* con la referencia a cada una de las clases de las que herede.

```
<packagedElement xmi:type="uml:Class" xmi:id="_19_0_2_b7002e9_1623259140788_209921_4834" name="A" visibility="public" isAbstract="true">
  <generalization xmi:id="_19_0_2_b7002e9_1623259147076_722508_4861" general="_19_0_2_b7002e9_1623259125100_267447_4808"/>
</packagedElement>
```

Figura 7. Representación Eclipse UML2 XMI para Clase.

Elemento 4. Atributo.

En USE los atributos se representan con el nombre del mismo seguido de dos puntos y su tipo. Estos se añaden uno debajo del otro después de la cabecera *attributes* de su clase. Ni su nombre ni su tipo pueden estar vacíos. El tipo puede ser un uno de los primitivos, un enumerado, una clase/clase de asociación o una colección (*Bag*, *Set* o *Sequence*) de cualquiera de estos elementos. Además, pueden estar derivados o inicializados con una expresión OCL. Los tipos primitivos de USE son *String*, *Integer*, *Real* y *Boolean*.

```
attString : String
attInteger : Integer init: 16
attBoolean : Boolean init : self.attInt.oclIsUndefined()
attboolean : Boolean derive : self.attInt.oclIsUndefined()
attClass   : A
attSequenceEnum : Sequence(NombreEnumerado)
```

En MagicDraw los atributos se añaden a la clase correspondiente en el apartado de *Attributes*. El tipo y el nombre pueden estar vacíos. El tipo puede ser uno de sus múltiples primitivos, un enumerado o una clase o clase de asociación. Para inicializarlo se debe añadir la expresión al campo *Default Value* y si, además, se marca como verdadero el campo *Is Derived*, será un atributo derivado. Para indicar que el tipo es una colección se utilizan los campos *Multiplicity*, *Is Ordered* y *Is Unique*. Si la multiplicidad superior es mayor que 1, será una colección. A partir de ahí, si es ordenada será una secuencia, si es única un conjunto y, si no cumple ninguno, una bolsa.

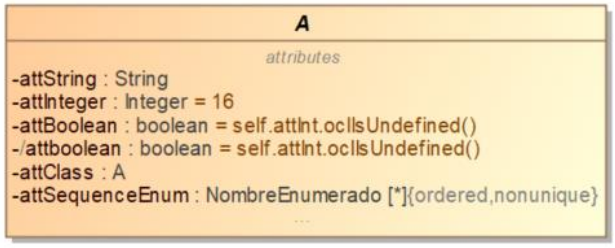


Figura 8. Representación MagicDraw para Atributo.

Su equivalencia en XMI es un elemento *ownedAttribute* que contiene en su cabecera el nombre del mismo y el tipo, aunque solo en el caso de que sea una referencia a un elemento existente. En caso de ser un tipo primitivo se añadirá dentro del objeto un elemento *type* que lo referencie. La inicialización y derivación funciona igual que en MagicDraw pero con el atributo *isDerived* de la cabecera y el elemento *defaultValue*. De igual forma, las colecciones de atributos se identifican con los campos *isOrdered*, *isUnique* y el valor del objeto *upperValue*.

Figura 9. Representación Eclipse UML2 XMI para Atributo.

Elemento 5. Operación.

En USE las operaciones se escriben después de la cabecera *operations* de su clase y consisten en el nombre obligatorio de la operación seguido de un paréntesis abierto, todos sus parámetros, si es que tiene, y un paréntesis de cierre. Los parámetros se representan en un formato idéntico al de los atributos, pero separados por comas. Asimismo, si la operación devuelve un objeto, se especifica su tipo poniendo previamente dos puntos. Los tipos permitidos son los mismos que para los atributos.

A partir de este punto, la operación será de consulta (*query*) si iguala a una expresión OCL. Si no lo es, se puede añadir el cuerpo de la operación en lenguaje SOIL entre los separadores *begin* y *end*.

Además, las operaciones pueden incluir precondiciones y postcondiciones, que se escribirán con las palabras pre y post, respectivamente, seguidos de un nombre opcional, dos puntos y una expresión OCL. El orden entre condiciones no es relevante. Además, estas también se podrán añadir al final del archivo, después del título *constraints* y del contexto de clase y operación a la que aplican.

```
class A
...
operations
nombreQuery() : Boolean = self.oclIsUndefined()
nombreOperacion(param1 : A, param2 : String)
      begin
            self.prueba := self.prueba + 4;
            self.prueba := self.prueba - 2;
            result := self.prueba
      end
      pre condicion1: true
...
end

constraints
context A :: nombreOperacion(param1 : A, param2 : String)
      post condicion2: self.oclIsUndefined()
```

En MagicDraw, las operaciones se definen dentro de la especificación de su clase en el apartado de *Operations*. Estas tienen un nombre, que puede ser vacío, e indican el tipo de elemento que devuelve, si es que lo hace, con el campo *Type*.

Asimismo, si la operación es de consulta se debe activar el campo *IsQuery* y, independientemente del tipo de operación, la expresión OCL o el código SOIL se añade en el campo *Body Condition*. En la misma pestaña, las condiciones solo pueden ser añadidas como elementos *Constraint*, que se explicarán posteriormente, dentro de los campos *Precondition* o *Postcondition*.

Por último, los parámetros se añaden en la sección *Parameters*, agregando un objeto *Parameter* al que se le puede asignar un nombre y un tipo.

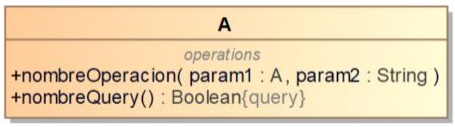


Figura 10. Representación MagicDraw para Operación.

El caso de XMI se estructura de manera similar al de MagicDraw. Las operaciones son elementos *ownedOperation* contenidas dentro de la clase que las posee. En su cabecera se indica su nombre, si son de consulta y las referencias a los objetos que representan su cuerpo de operación, sus precondiciones y sus postcondiciones. Estos tres últimos son elementos *ownedRule* que constan dentro de la operación.

De igual forma, la operación también aloja objetos *ownedParameter*, que se estructuran de forma similar a los atributos y representan los parámetros. El parámetro con dirección *return* define el tipo que devuelve la operación.

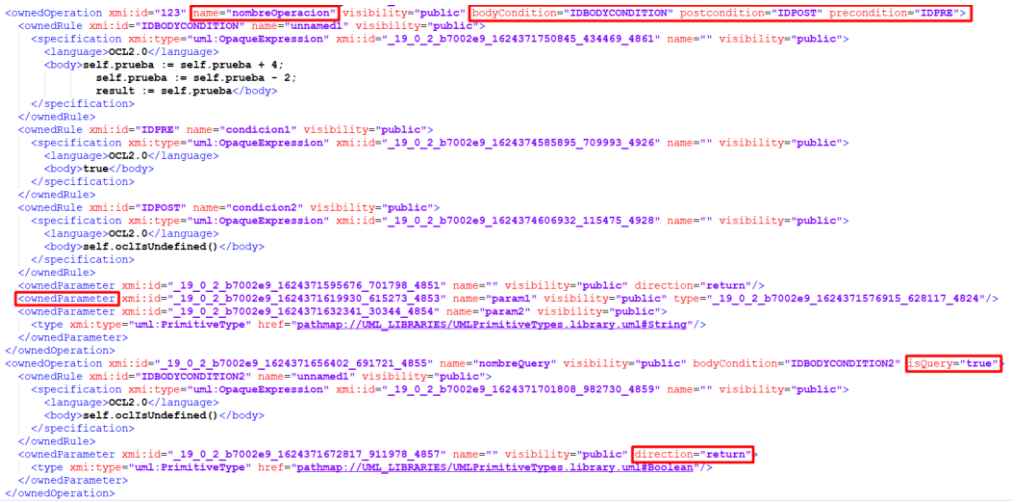


Figura 11. Representación Eclipse UML2 XMI de Operación.

Elemento 6. Restricción.

En USE las restricciones pueden añadirse tanto en la misma clase a la que restringen como al final del archivo tras el título *constraints*. En este segundo caso se deberá aclarar a que clase pertenece la invariante colocando previamente la palabra *context* seguida del nombre de la clase.

Las invariantes se declaran con la palabra *inv* seguida de un nombre, que en este caso es opcional, dos puntos y una expresión OCL obligatoria.

```
class A
constraints
        inv: self.oclIsUndefined()
end

constraints
context A
        inv restriccion: true
```

En el caso de MagicDraw, únicamente se pueden añadir restricciones en el apartado *Constraints* de la clase. Al agregar un nuevo elemento se le podrá asignar un nombre e incluir, si se desea, su expresión OCL en el campo de *Specification*.

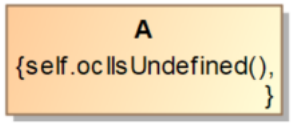


Figura 12. Representación MagicDraw para Restricción.

Tras exportarse a XMI, la restricción se representa con un elemento *OwnedRule* dentro de la clase a la que aplica, aunque la vuelve a referenciar mediante el campo *constrainedElement* de su cabecera. Contiene, a su vez, un objeto *specification* que indica el lenguaje y el cuerpo de la expresión.

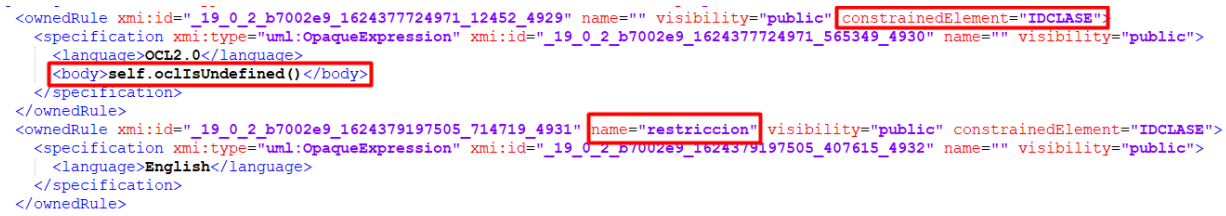


Figura 13. Representación Eclipse UML2 XMI para Restricción.

Elemento 7. Máquina de estados.

En USE las máquinas de estados de protocolo se incluyen dentro de la clase a la que se refieren, al igual que los atributos y las operaciones. Una clase puede contener más de una máquina de estados y estas se inicializan con la palabra *psm* seguida de el nombre de la misma. Posteriormente, hay dos cabeceras obligatorias y cuyo orden es estricto: *states* y *transitions*.

En el apartado *states* se incluye el nombre de todos los estados de la máquina, con un mínimo de dos. De forma opcional, cada estado puede estar restringido por una invariante, cuya representación es una expresión OCL entre corchetes.

Además, es obligatoria la existencia de un único estado inicial, que se marcará colocando dos puntos y la palabra *initial* después del nombre. Por otro lado, se pueden tener ninguno, uno o varios estados finales, que se indicarán igual pero utilizando la palabra *final*.

Para que la máquina sea válida, mínimo debe tener una transición, que no se permite que sea del estado inicial a sí mismo. Las transiciones se indican separando con una flecha hacia la derecha el estado fuente del estado objetivo. Opcionalmente, puede añadirse el nombre de la operación que dispara la transición, colocándolo entre llaves. De igual forma, también se puede añadir una precondición y/o una postcondición a la operación, colocando la expresión OCL correspondiente antes o después su nombre.

```
psm maquinaestados
states
	s : initial
	estado0	[self.pruebaatributo > 0]
	estado1
	estado2	[self.pruebaatributo < 20]
	estado3
	f : final
transitions
	s -> estado0
	estado0 -> estado1 { pruebaoperacion() }
	estado1 -> estado2
	estado2 -> f { [self.pruebaatributo < 20] pruebaoperacion()}
	estado2 -> estado3  { [self.pruebaatributo >= 20] pruebaoperacion()}
	estado3 -> estado3  { pruebaoperacion() [self.pruebaatributo < 10]}
	estado3 -> f { pruebaoperacion() [self.pruebaatributo >= 10]}
end
```

Por su lado, en MagicDraw se debe crear una *Protocol State Machine* sobre la clase que se desee aplicar. La creación de este elemento generará un modelo específico de la máquina, donde se podrán añadir, entre otros, elementos *State* para los estados normales, *Initial* para el inicial y *Final State* para los finales. Si el estado cuenta con una invariante, esta se añade en el campo *State Invariant*.

Por otro lado, para las transiciones se establecen relaciones *Protocol Transition* entre dos estados. A esa relaciones se les puede definir los campos *Operation*, que permite elegir entre las operaciones disponibles en la clase, *Pre Condition* y *Post Condition*.

Este modelo no tiene mayores restricciones que la existencia de, como máximo, un único estado inicial. Los estados pueden no tener nombre y, si no son del mismo tipo, pueden tenerlo repetido.

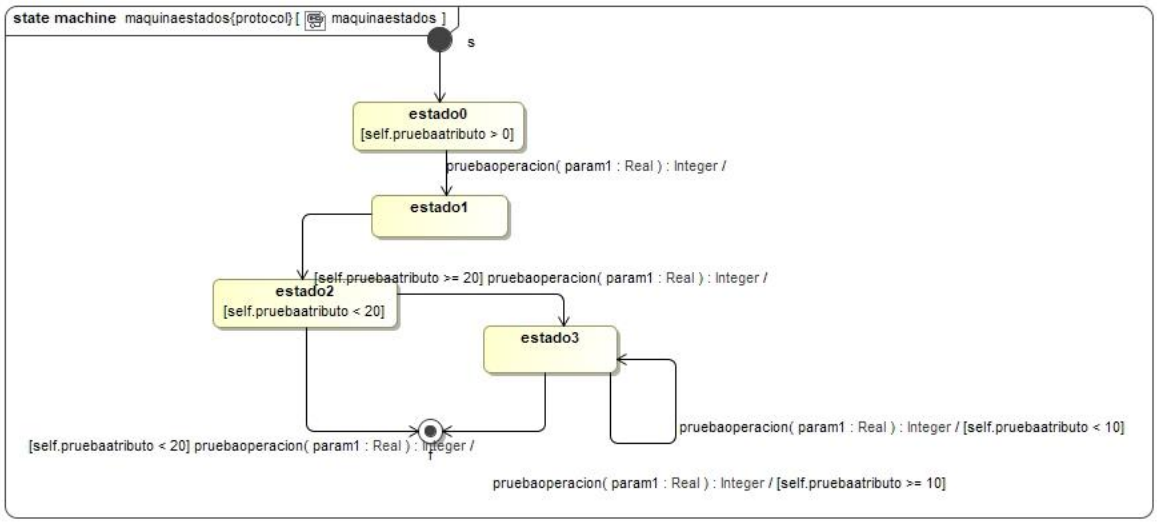


Figura 14. Representación MagicDraw de Máquina de estados.

Al exportarlo en XMI, se obtiene un elemento *ownedBehavior* empaquetado en la clase correspondiente. Dentro hay un objeto *region* y en este se encuentran tanto las transiciones en objetos *transition* como los estados en *subvertex*.

Empezando por los estados, estos varían su tipo dependiendo de si es final (*uml:FinalState*), inicial (*uml:Pseudostate*) o un estado simple (*uml:State*). Si este contase con una invariante, entonces contendría su respectivo *ownedRule*, al que se referenciaría en el campo *stateInvariant* de la cabecera.

Las transiciones, por otro lado, referencian a sus estados fuente y objetivo en sus campos *source* y *target*, respectivamente. En el mismo lugar se referencian las condiciones, si las hay, utilizando los campos *preCondition* y *guard* en el caso de las precondiciones y el campo *postCondition* en el de las postcondiciones. Para indicar la operación que la dispara, se añade un elemento *trigger* con una referencia a un objeto *event* externo a la clase. Este es un elemento empaquetado de tipo *uml:CallEvent* y contiene el identificador de la operación.

Figura 15. Representación en Eclipse UML2 XMI para Máquina de estados.

Elemento 8. Asociación.

En USE las asociaciones comienzan con la palabra *association*, *aggregation* o *composition*, dependiendo si es una relación normal, de agregación o de composición, respectivamente. Lo siguiente es el nombre obligatorio de la relación y la palabra *between*. A partir de aquí se listan los miembros finales de la asociación y se termina con un *end*.

Los miembros finales se representan con el nombre de la clase seguido de unos corchetes que contienen la multiplicidad de ese lado. Opcionalmente también se puede añadir la palabra *role* seguida del nombre del rol y, si la multiplicidad es mayor que 1, se puede establecer la colección como ordenada con la palabra *ordered*. Aunque establecer un rol es optativo, en caso de no establecerlo USE asumirá que el rol es el nombre de la clase en minúsculas. Esto únicamente genera conflicto cuando existen al menos dos asociaciones entre las mismas clases (o misma en caso de que sea reflexiva), ya que el rol actúa como un atributo y estos no pueden tener nombres repetidos.

Cabe destacar que en esta gramática no existe la no navegabilidad en las relaciones.

```
association AS between
      A [0..1] role ejemplo
      B [1..5] role ejemplo1
end

composition CO between
      A [0..*]
      B [1..*]
end

aggregation AG between
      A [*] role AG1
      B [3] role AG2
end
```

En cuanto a MagicDraw, la herramienta te permite seleccionar entre relaciones de tipo *Association*, *Composition* y *Aggregation*, entre otras con las que no cuenta USE, y asignar las dos clases extremo. Luego, en su especificación, se puede asignar un nombre y, si se desea, definir en el apartado de *Roles* la multiplicidad (*Multiplicity*), el rol (*Name*), la navegabilidad (*Navigable*) de cada lado y, en el caso de que la multiplicidad sea mayor a 1, si la colección es ordenada (*Is Ordered).*

Para crear relaciones no binarias se puede añadir un elemento *N-ary Association*, que en realidad actuará exactamente igual que una clase.

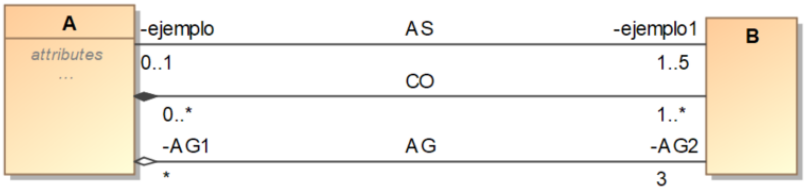


Figura 16. Representación MagicDraw para Asociación.

En el XMI equivalente, las asociaciones son elementos empaquetados en el modelo de tipo *uml:Association* que cuentan con un campo para el nombre de la misma y otro campo *memberEnd* que especifica, separados por espacios, las referencias de los atributos que representan sus extremos de asociación.

Estos atributos están contenidos cada uno en la clase que les corresponde, con la misma estructura que los atributos ya explicados, aunque el nombre en este caso es el rol del extremo. Además, se añade en la cabecera un campo *association* que lo relaciona con su asociación, un campo *isOrdered* si su multiplicidad es mayor a 1 y la colección está ordenada y, si no es una asociación simple, un campo *aggregation,* cuyo valor indica si es una agregación (*shared*) o una composición (*composite*).

```
<packagedElement xmi:type="uml:Class" xmi:id="_19_0_2_b7002e9_1623267664200_962627_4823" name="A" visibility="public">
  ...
  <ownedAttribute xmi:id="IDMECO1" name="" visibility="private" type="_19_0_2_b7002e9_1623267822449_572478_4931" aggregation="composite" association="IDCOMPOSICION">
    <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_19_0_2_b7002e9_1623268386456_402063_5136" name="" visibility="public" value="1"/>
    <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_19_0_2_b7002e9_1623268386456_450378_5137" name="" visibility="public" value="*"/>
  </ownedAttribute>
  <ownedAttribute xmi:id="IDMEAG1" name="AG2" visibility="private" type="_19_0_2_b7002e9_1623267822449_572478_4931" aggregation="shared" association="IDAGREGACION">
    <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_19_0_2_b7002e9_1623268414010_703314_5144" name="" visibility="public" value="3"/>
    <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_19_0_2_b7002e9_1623268414010_514236_5145" name="" visibility="public" value="3"/>
  </ownedAttribute>
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="_19_0_2_b7002e9_1623267822449_572478_4931" name="B" visibility="public">
  ...
  <ownedAttribute xmi:id="IDMECO2" name="" visibility="private" type="_19_0_2_b7002e9_1623267664200_962627_4823" association="IDCOMPOSICION">
    <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_19_0_2_b7002e9_1623268383737_64000_5132" name="" visibility="public"/>
    <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_19_0_2_b7002e9_1623268383738_367276_5133" name="" visibility="public" value="*"/>
  </ownedAttribute>
  <ownedAttribute xmi:id="IDMEAG2" name="AG1" visibility="private" type="_19_0_2_b7002e9_1623267664200_962627_4823" association="IDAGREGACION">
    <lowerValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_19_0_2_b7002e9_1623268417427_331038_5148" name="" visibility="public"/>
    <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_19_0_2_b7002e9_1623268417427_541020_5149" name="" visibility="public" value="*"/>
  </ownedAttribute>
</packagedElement>
<packagedElement xmi:type="uml:Association" xmi:id="IDCOMPOSICION" name="CO" visibility="public" memberEnd="IDMECO1 IDMECO2"/>
<packagedElement xmi:type="uml:Association" xmi:id="IDAGREGACION" name="AG" visibility="public" memberEnd="IDMEAG1 IDMEAG2"/>
<packagedElement xmi:type="uml:Class" xmi:id="_19_0_2_b7002e9_1623272640377_257729_5154" name="" visibility="public"/>
```

Figura 17. Representación Eclipse UML2 XMI para Asociación.

Elemento 9. Clases de asociación.

Por último, en las tres gramáticas, las clases de asociación son una mezcla entre la clase y la asociación, como su nombre bien indica.

En USE comienza igual que una asociación pero utilizando la palabra *associationclass* y, tras la lista de miembros finales, termina teniendo los mismos apartados que cualquier clase.

```
associationclass C between
        A [1]
        B [1]
attributes
        ejemplo : Real
end
```

En MagicDraw se crea con el elemento *Association Class* y, tras haber asignado sus dos extremos, funciona exactamente igual que el resto de clases.

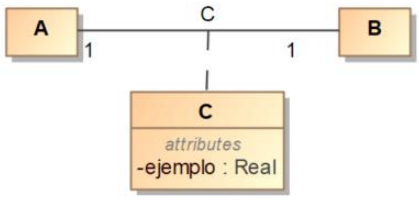


Figura 18. Representación MagicDraw para Clase de asociación.

En XMI son prácticamente idénticas a las clases pero de tipo *uml:AssociationClass* y con el campo *memberEnd*, propio de las asociaciones, en su cabecera.

```
<packagedElement xmi:type="uml:AssociationClass" xmi:id="IDASSOCIATIONCLASS" name="C" visibility="public" memberEnd="IDME1 IDME2">
  <ownedAttribute xmi:id="_19_0_2_b7002e9_1624395365525_993472_5085" name="ejemplo" visibility="private">
    <type xmi:type="uml:PrimitiveType" href="pathmap://UML_LIBRARIES/JavaPrimitiveTypes.library.uml#double"/>
  </ownedAttribute>
</packagedElement>
```

Figura 19. Representación Eclipse UML2 XMI para Clase de asociación.

4.2. Diferencias remediables

A modo de resumen, se listarán las diferencias encontradas durante el análisis cuya falta de equivalencia directa se puede solucionar fácilmente, logrando un modelo mucho más fiel y asegurando la correcta lectura del archivo generado.

Diferencia 1. En USE los únicos elementos que pueden no tener un nombre específico son las restricciones (invariantes, precondiciones y postcondiciones), a diferencia de MagicDraw que sí permite dejar el nombre vacío en cualquier elemento. Esto se soluciona estableciendo como nombre *unnamed* en la conversión a USE.

Diferencia 2. MagicDraw permite repetir nombres de elementos que no son exactamente del mismo tipo, mientras que en USE los enumerados, clases y asociaciones no pueden tener nombres coincidentes. Lo mismo pasa con el resto de elementos dentro de su contexto. Esto se puede remediar fácilmente concatenando un uno al nombre repetido e incrementándolo si se encuentran más.

Diferencia 3. Algo parecido a las dos anteriores diferencias ocurre con el tema de los roles de asociación. En USE no hay problemas al no especificar los roles siempre que varios con el mismo nombre no coincidan en ir hacia la misma clase. Esto también es un problema si el campo es directamente vacío, ya que para USE eso es equivalente a poner de rol el nombre de la clase en minúsculas. La solución a esto es la misma que en el punto anterior, pero teniendo en cuenta que se deben comprobar respecto al resto de miembros de asociación que recibe como atributos la clase contraria.

Diferencia 4. En USE las restricciones están obligadas a contener una expresión OCL, mientras que en MagicDraw no es obligatorio rellenar ese campo. Esto se resuelve asignando *true* a toda aquella restricción que no contenga especificación o sea vacía.

Diferencia 5. Al igual que en la anterior diferencia, en USE los enumerados deben contener al menos un valor, mientras que en MagicDraw pueden no tener ninguno. La solución es asignar un valor por defecto *requiredValue* a la lista de valores del enumerado.

Diferencia 6. De la misma manera, el tipo de un atributo o parámetro no puedo ser indefinido en USE, aunque si en MagicDraw. Por ello, en la conversión a USE, se asignará el tipo *String* a todo atributo con tipo indefinido o nulo.

Diferencia 7. MagicDraw cuenta con un abanico mucho más amplio de tipos primitivos, mientras que en USE solo existen los cuatro antes mencionados. Por ello, exceptuando los que sí existen, se establecerán los tipos *float* y *double* como *Real*, *short* y *long* como *Integer* y cualquier otro tipo como *String*.

Diferencia 8. MagicDraw permite las relaciones con un extremo no navegable, las cuales no existen en la gramática de USE. Así, todas las relaciones de este tipo

son sustituidas en la conversión a USE por un atributo en la clase no navegable, ya que conceptualmente no hay diferencias entre un atributo que referencia a una clase y una relación que solo es navegable en un sentido.

Diferencia 9. En USE es necesario que una máquina de estados contenga mínimo 2 estados y una transición, mientras que en MagicDraw no existen estos requerimientos. En el caso de encontrar este hecho, se escribiría la máquina como una más, pero comentándola con sus respectivos dobles guiones.

4.3. Limitaciones

De igual forma, las características que se han limitado de ambas gramáticas por su nula o compleja equivalencia son las siguientes:

Limitación 1. La falta de detalle en los modelos de USE incapacita la conversión de la gran cantidad de información sobre el modelo que es capaz de almacenar MagicDraw.

Limitación 2. Debido al diseño de las relaciones n-arias en Eclipse UML2 XMI, es complejo generar asociaciones de ese estilo a partir de USE, por lo que se ha optado por considerarlas todas las de este lenguaje como binarias. Esto afecta tanto a los miembros de la asociación como a sus multiplicidades. De estas últimas se tienen en cuenta solo las primeras de la lista.

Limitación 3. Aunque es posible, adaptar máquinas de estados con estados compuestos o de submáquinas supone un gran esfuerzo. Esto es debido a que la gramática de USE no los permite, por lo que habría que adaptar la máquina de tal forma que solo quedase la principal con todos los estados y transiciones contenidos bien conectados entre sí.

Limitación 4. USE permite que las colecciones sean de más de un tipo, mientras que MagicDraw solo permite asignar un único tipo. Por ello, si se recibe una colección de este tipo, solo se considerará el primero de ellos.

Finalmente, a raíz de todo lo expuesto en este capítulo, sumado a otras características que son lógicas en cualquier programa similar, se especificaron los requisitos funcionales y no funcionales de la aplicación, que se encuentran anexados en el apéndice E.

5

Diseño de la herramienta

A continuación, se expondrán los diseños finales de la herramienta, obtenidos tras haber pasado por todas las iteraciones. Estos ayudarán al lector a comprender las decisiones de diseño tomadas durante el desarrollo, documentando su arquitectura, implementación y funcionamiento.

5.1. Diagrama de componentes

Como se puede ver en la Figura 20, finalmente la herramienta se ha desarrollado en dos paquetes o proyectos distintos.

El denominado *modelConverter.use_language* es generado automáticamente por el plugin de Xtext y es el principal responsable de la conversión de USE a XMI.

Por otro lado, el paquete *modelConverter* se encarga de la conversión de XMI a USE mediante el plugin de UML2, de unificar ambas conversiones en una única interfaz y de ejecutar las pruebas JUnit5 de la aplicación.

Además, ambos proyectos utilizan el plugin de EMF, ya que tanto Xtext como UML2 dependen de él.

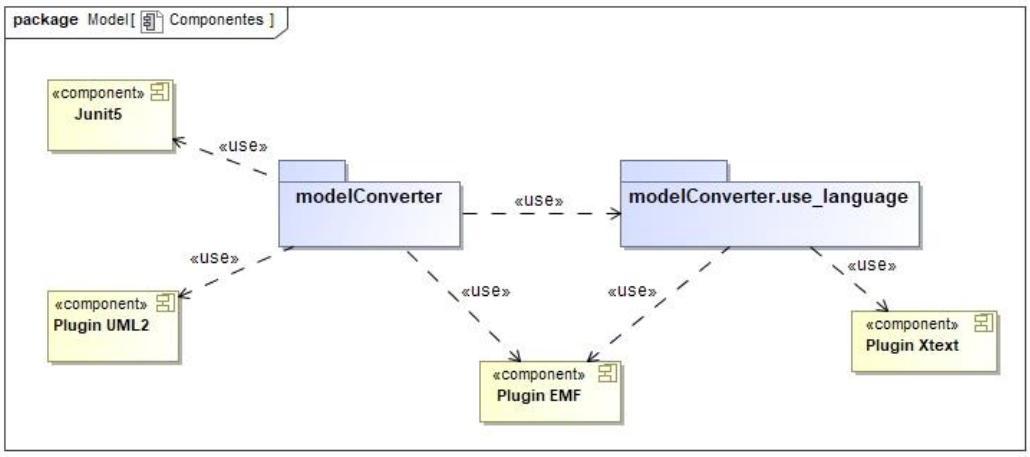


Figura 20. Modelo de componentes.

5.2. Diagrama de clases

En este apartado se detallará casi exclusivamente el paquete *modelConverter*, ya que el proyecto de Xtext se ha generado automáticamente y no ha necesitado ser diseñado.

La arquitectura del proyecto corresponde con el patrón modelo-vista-controlador (MVC), que se caracteriza por separar la interfaz de usuario de la lógica de negocio, relacionándola únicamente mediante controladores. Este patrón favorece el mantenimiento y la reutilización del código.

En la Figura 21 se puede ver un diagrama simplificado de la arquitectura, sin tener en cuenta atributos y operaciones.

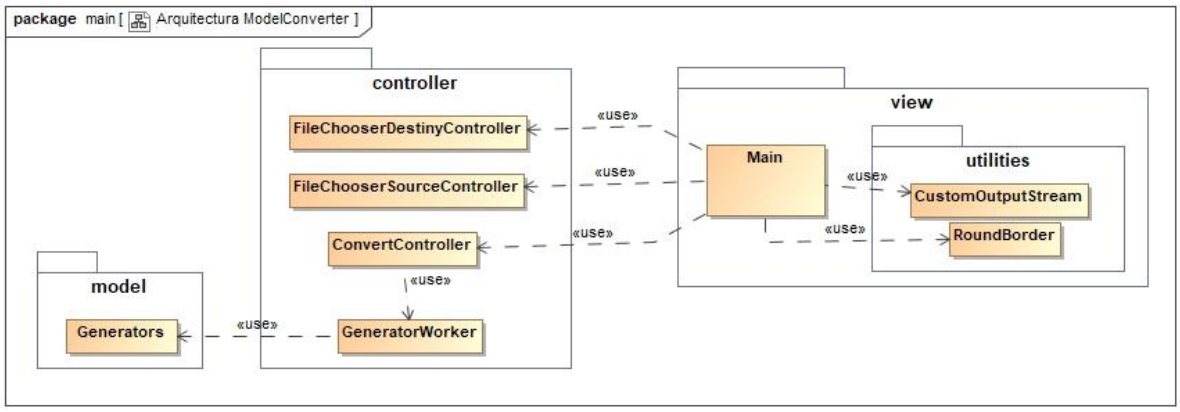


Figura 21. Arquitectura del componente *modelConverter*

Podemos ver que realmente la interfaz de la herramienta es muy sencilla, ya que consta de una única vista. Esta cuenta con un controlador para cada uno de sus botones. Además, para no congelar el programa, se ha decido ejecutar el modelo a través de una hebra, creada e iniciada al pulsar el botón de convertir.

Los siguientes diagramas de clases representan, con más detalle, la parte correspondiente al modelo de la herramienta. La Figura 22 no tiene en cuenta las operaciones para facilitar su comprensión, mientras que en el apéndice C se muestra el diagrama completo.

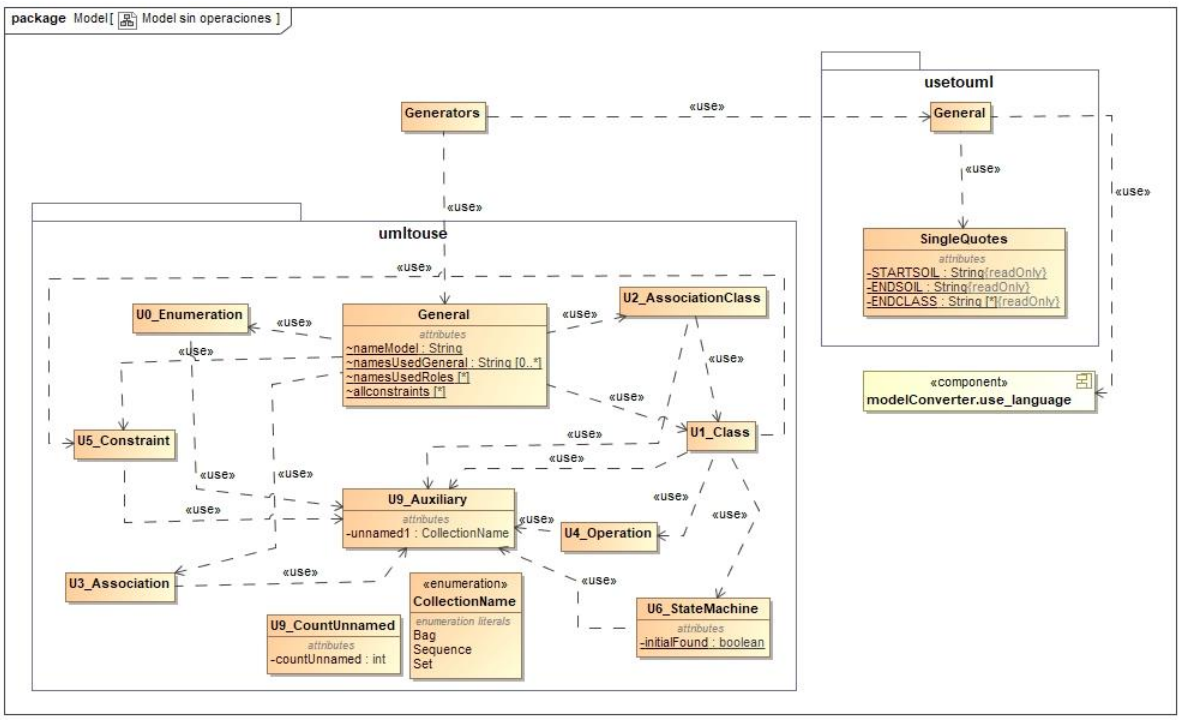


Figura 22. Diagrama de clases del modelo de la herramienta sin operaciones

Una de las principales decisiones de diseño fue la de elegir el pivote de la herramienta. Aunque se podría haber creado un nuevo formato intermedio, finalmente se decidió utilizar el formato Eclipse UML2 XMI, cuyo sufijo es .uml, aprovechando que es un estándar en el intercambio de modelos y que muchas herramientas visuales, como MagicDraw, lo importan y exportan.

Otra de las decisiones que se tomaron fue la de tratar la mayoría de los atributos y las operaciones como estáticos. Esto se debe a que no es necesario un diseño orientado a objetos para cumplir la funcionalidad requerida, por lo que en su lugar se han utilizado clases estáticas. Como Java no permite de forma directa estas clases, se han privatizado los constructores para evitar su instanciación.

Respecto al diagrama, la clase principal del modelo es *Generators* que, dependiendo del sufijo del archivo que reciba, enviará los datos de entrada a uno de los dos subpaquetes.

El subpaquete responsable de transformar .use a .uml proporciona la ruta del fichero fuente al método de *SingleQuotes*, que coloca comillas simples antes y después del cuerpo de las operaciones de USE (entre los separadores *begin* y *end)* existentes y devuelve un archivo temporal. Posteriormente, este archivo es procesado por el componente de *modelConverter.use_language,* generando el archivo .uml correspondiente.

Poner las comillas permite que el código SOIL pueda ser procesado como tipo texto por Xtext, ya que no se ha añadido la gramática correspondiente a este otro lenguaje. Respecto al proyecto de Xtext, hay que destacar los archivos *USE.xtext*, que contiene la gramática de USE, y *USEGenerator.xtend*, que recorre los objetos EMF generados por Xtext creando el fichero .uml.

Por otro lado, el subpaquete encargado de transformar .uml a .use obtiene los objetos EMF a partir del plugin UML2 y los va recorriendo por elementos, escribiendo el código USE correspondiente en un objeto *StringBuilder* [31]. El contenido de este objeto se traspasará finalmente a un nuevo fichero .use.

Siguiendo este patrón, se ha creado una clase para cada tipo de elemento UML que necesite ser transformado a USE. En el método principal de estas clases se recorren todos los componentes del paquete que sean de dicho elemento, utilizando los métodos auxiliares para las comprobaciones comunes.

5.3. Diagramas de secuencia

Por último, se detallará el funcionamiento de la aplicación y su interacción con el usuario por medio de una serie de diagramas de secuencia que recogen todas las posibles entradas. Estos se encuentran en las páginas siguientes.

Caso 1. Ejecución correcta: el usuario introduce ruta de fichero fuente y ruta destino correctas. El fichero es .use o .uml y es gramaticalmente correcto.

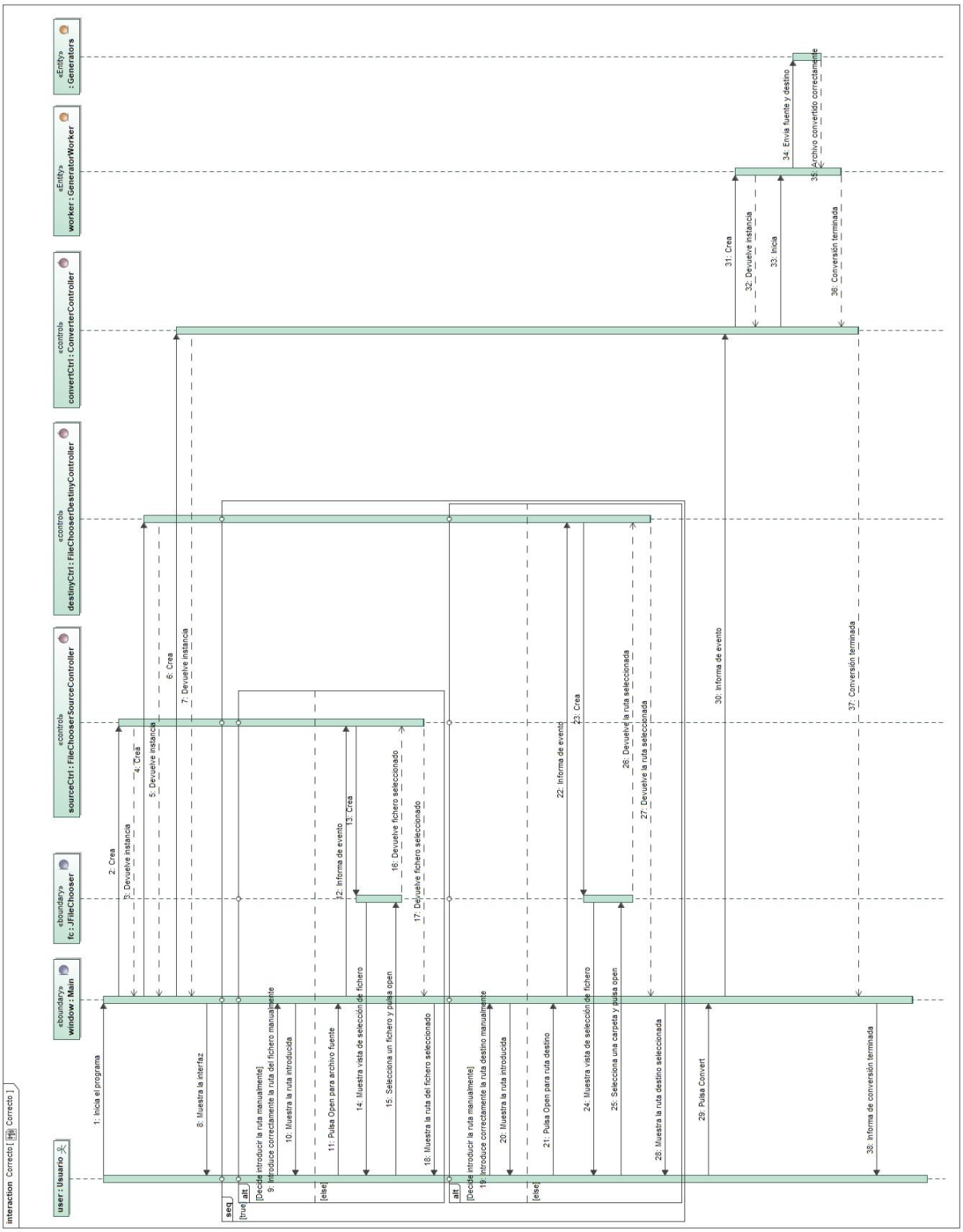


Figura 23. Diagrama de secuencia para ejecución correcta.

Caso 2. Ejecución incorrecta: el usuario deja vacío uno de los campos o los dos.

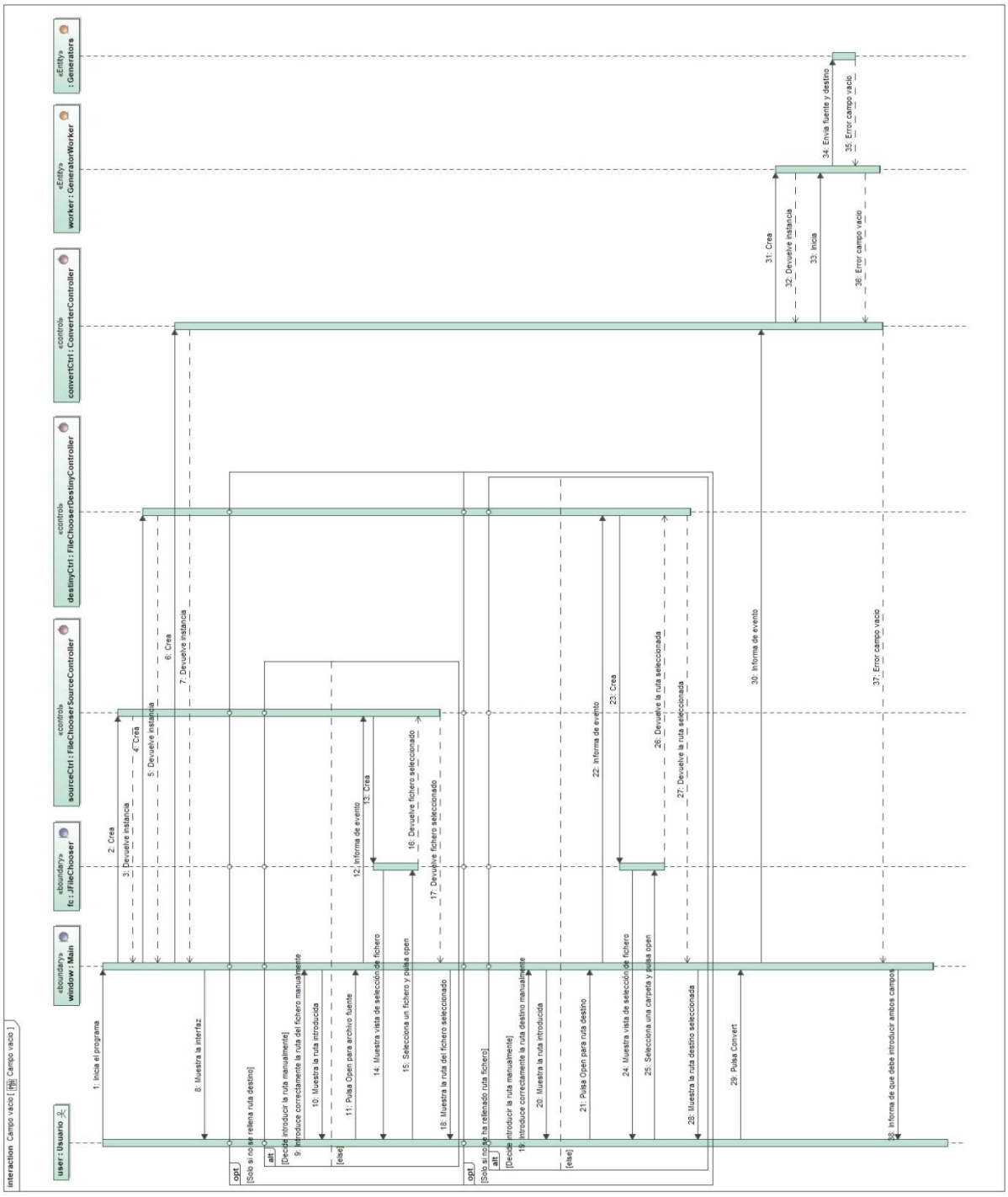


Figura 24. Diagrama de secuencia para ejecución incorrecta: campo vacío.

Caso 3. Ejecución incorrecta: la ruta del fichero fuente no es válida.

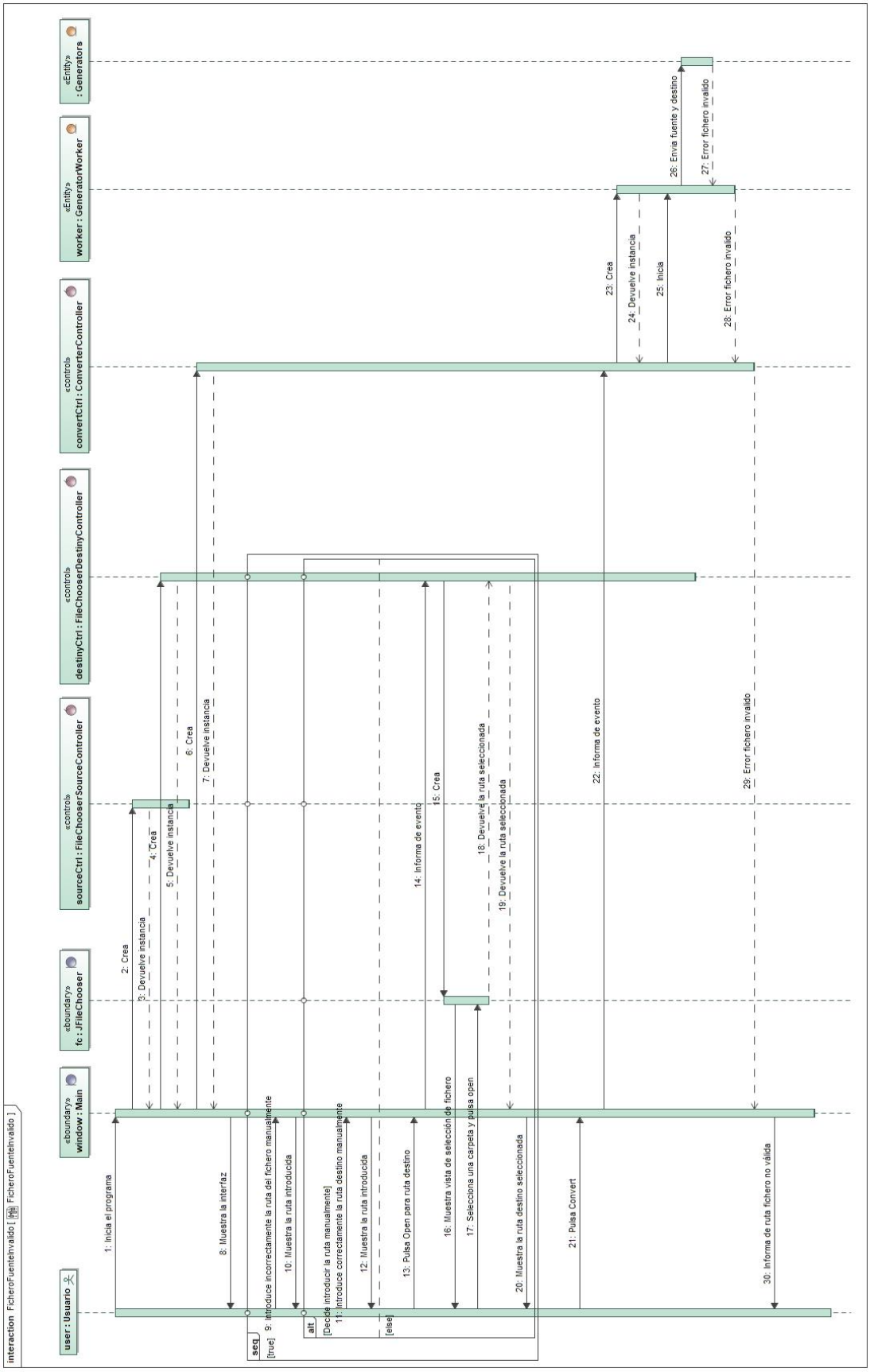


Figura 25. Diagrama de secuencia para ejecución incorrecta: ruta fichero inválida

Caso 4. Ejecución incorrecta: la ruta de destino no es válida.

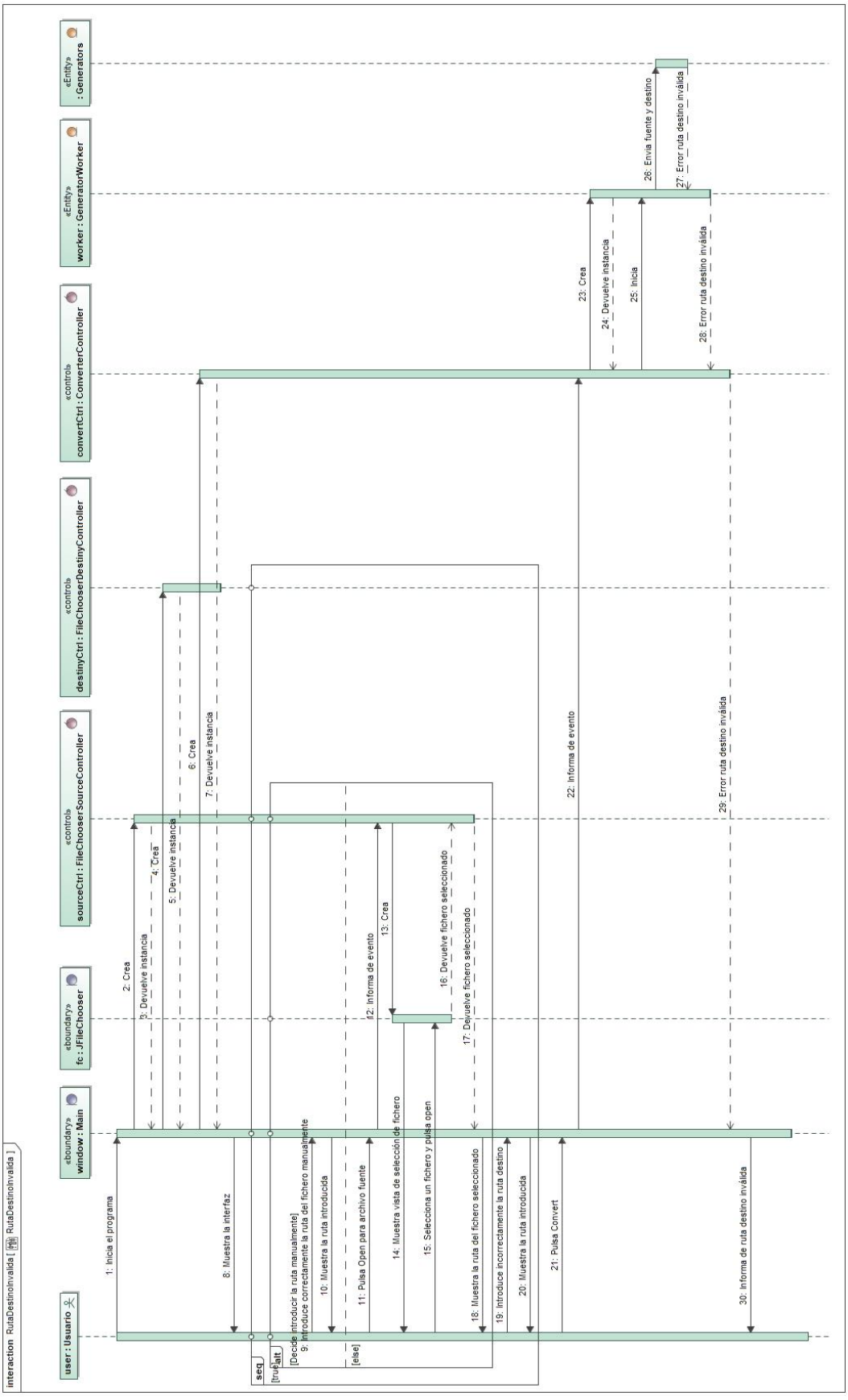


Figura 26. Diagrama de secuencia para ejecución incorrecta: ruta destino inválida

Caso 5. Ejecución incorrecta: sufijo de archivo no válido.

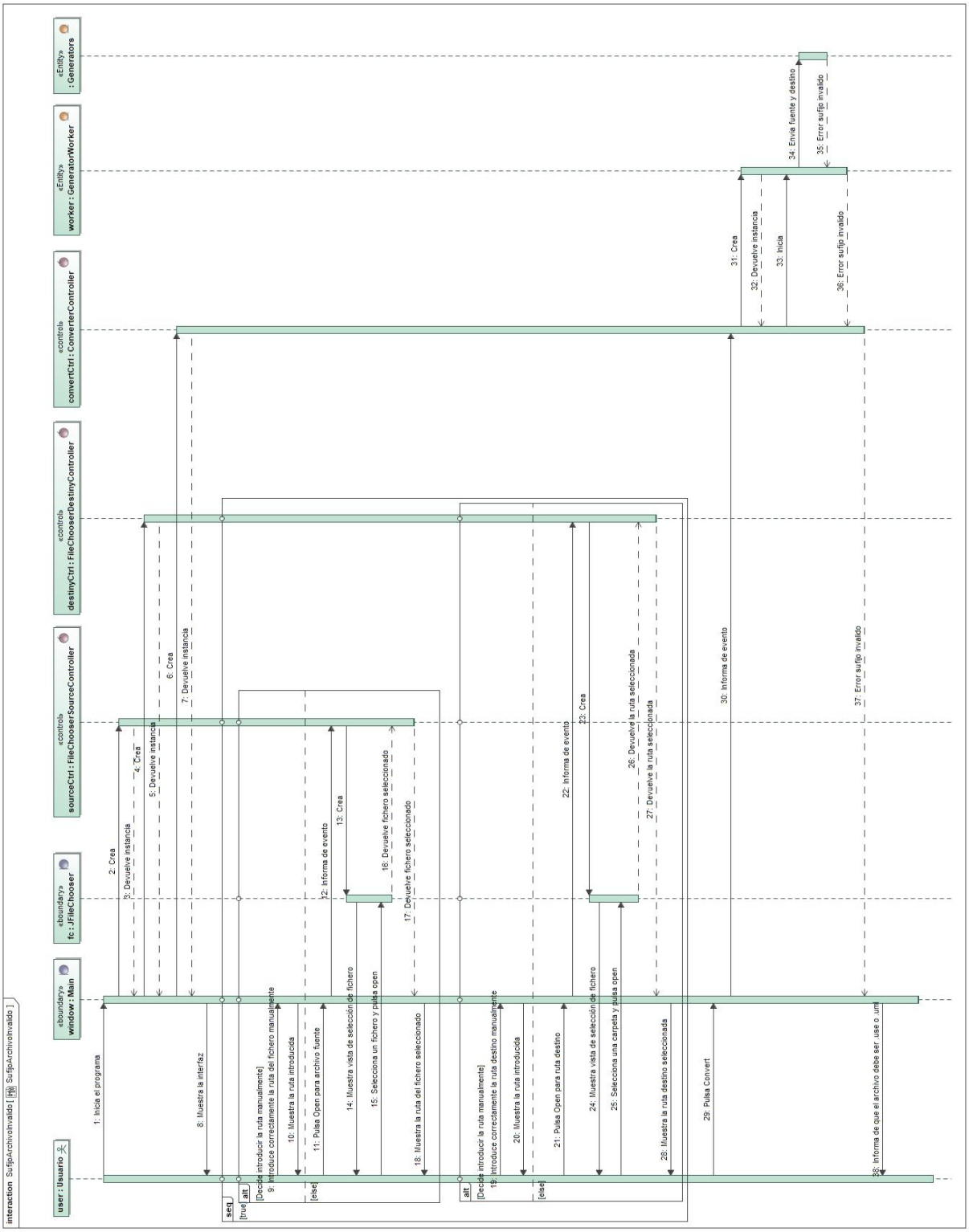


Figura 27. Diagrama de secuencia para ejecución incorrecta: sufijo de archivo inválido

Caso 6. Ejecución incorrecta: errores gramaticales en el archivo.

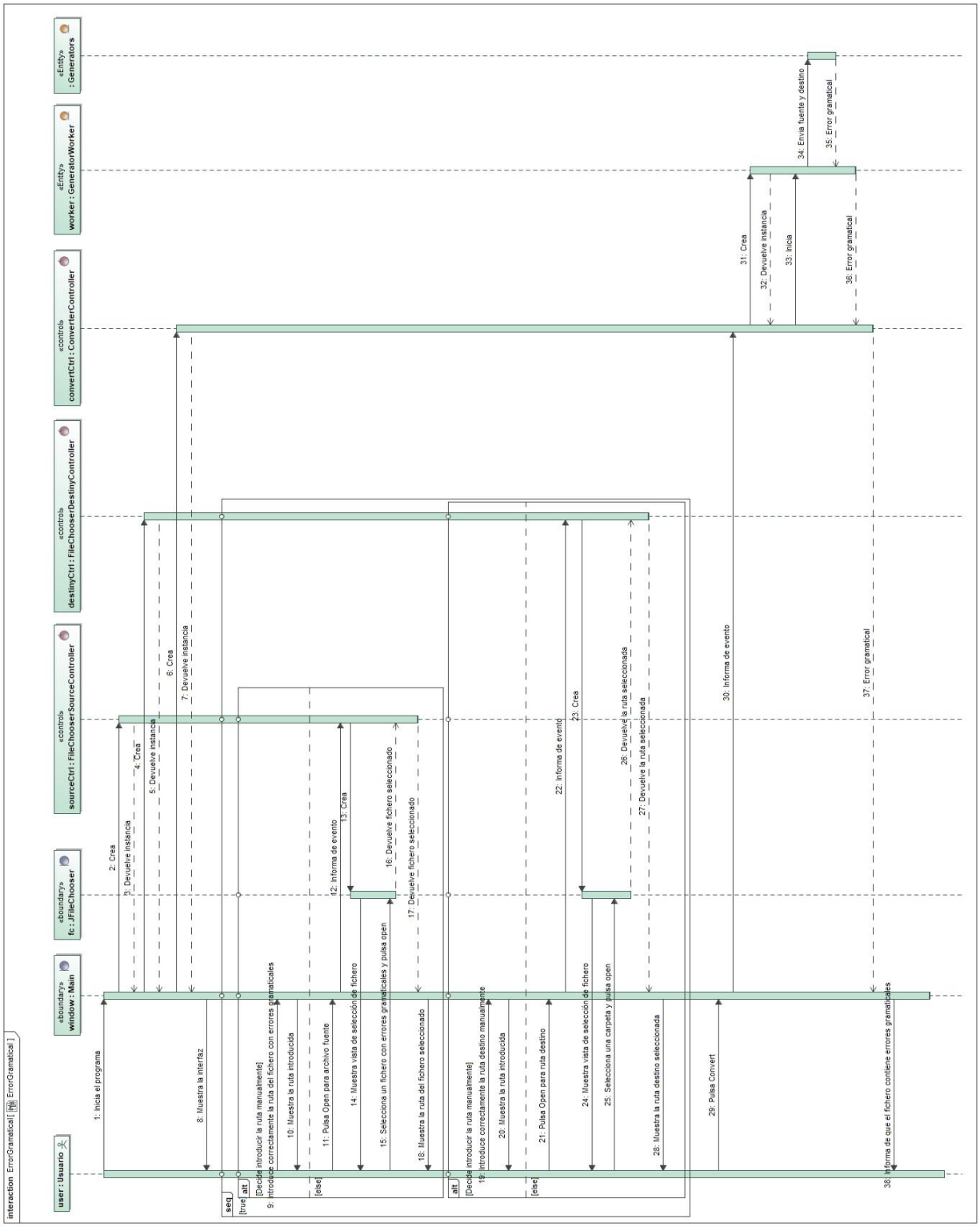


Figura 28. Diagrama de secuencia para ejecución incorrecta: error gramatical en archivo

6

Implementación

En este capítulo se comentará el proceso de implementación de la herramienta, junto con los problemas que han ido surgiendo y cómo se han solucionado.

Antes de comenzar, es importante destacar que tanto la implementación como los proyectos de prueba se salvaguardaron en todo momento en GitHub [32], una conocida plataforma de desarrollo que cuenta con control de versiones Git [33], cuyo enlace es: https://github.com/juliarobles/model-converter.

Además, tras finalizar completamente el desarrollo, la herramienta se ha exportado como archivo JAR ejecutable, permitiendo el intercambio de modelos de forma autónoma (*stand-alone*).

6.1. Conversión de UML a USE

Previa a la implementación de la herramienta, se siguió un tutorial para familiarizarse con EMF [34] y se realizó un pequeño programa para probar el

plugin de UML2. Este consistía en cargar el recurso a partir de un fichero e ir escribiendo por pantalla la información más relevante.

Posteriormente, se procedió con la implementación de la conversión, siguiendo el diseño expuesto anteriormente. A modo de resumen, el código obtiene la información necesaria de cada objeto mediante algunos de sus métodos y, mediante concatenaciones, devuelve el texto correspondiente a la transformación en lenguaje USE de ese elemento, siguiendo en todo momento la gramática expuesta en su documentación [35]. Además, previamente se realizan varias comprobaciones para verificar que se cumplen las limitaciones de USE frente a MagicDraw antes mencionadas (elementos sin nombres, enumerados sin valores, restricciones sin OCL, etc.).

Estos textos se concatenan unos con otros hasta obtener el código USE completo en un objeto *StringBuilder*. Finalmente, se creará un nuevo fichero .use en la carpeta de destino seleccionada y se escribirá todo el texto obtenido. Este archivo llevará el nombre modelConverter_X, siendo X el nombre del modelo. En el caso de que el fichero ya exista, se le añadirá un (1), (2), (3), etc. en lugar de sobrescribirlo.

Por otro lado, el principal problema que se tuvo en este apartado fue la escasa documentación del plugin y su antigüedad. A pesar de que esto último podría considerarse una ventaja, realmente este plugin no cuenta con una gran comunidad y en algunos casos ha sido complicado encontrar respuesta a lo que se estaba buscando, ya que muchas preguntas de foros no terminaban de dar una solución o enlazaban con páginas no existentes.

Teniendo en cuenta esto, la carga del recurso no fue tarea fácil, ya que, para permitir la ejecución autónoma del programa (*stand-alone*), su obtención se debía

configurar de una forma concreta que permitiese cargar correctamente todas las librerías dependientes.

De igual forma, los tipos primitivos de MagicDraw supusieron otra complicación, debido a que no se cargaban correctamente y aparecían como nulos. Finalmente se arregló añadiendo al proyecto un archivo extraído de MagicDraw que contenía la definición del perfil estándar de UML para la herramienta.

El resto de la implementación no supuso problemas mayores que el de encontrar en cada clase de UML2 los métodos que devolvían la información necesaria.

6.2. Conversión de USE a UML

De mismo modo, para implementar esta transformación se comenzó siguiendo los tutoriales recomendados por Xtext [36] [37]. Seguidamente, se creó un nuevo proyecto Xtext donde se escribió la gramática de USE siguiendo al pie de la letra su documentación [35], ya que la herramienta tenía que ser capaz de leer cualquier archivo USE aunque limitase ciertos aspectos ya comentados.

Una vez la gramática estuvo completa y funcionando correctamente, se procedió a rellenar el archivo Xtend correspondiente al generador. Este recorre los objetos EMF que Xtext genera automáticamente al ejecutar la gramática, añadiendo por medio todo el texto necesario para convertirlo a formato XMI. Este texto se obtuvo examinando archivos Eclipse UML2 XMI de modelos de prueba exportados desde MagicDraw. Destacar también que se ha utilizado el identificador único de cada objeto (*hashcode*) como identificador de su correspondiente elemento XMI.

Posteriormente, en la quinta iteración, la gramática y el generador fueron ampliados con lo respectivo a las máquinas de estados, siguiendo, de igual forma,

la documentación de USE correspondiente [38] y los modelos de prueba exportados a XMI.

Por otro lado, en el proyecto *modelConverter*, se creó un método capaz de utilizar el proyecto Xtext de forma que se pudiese cargar el fichero fuente como recurso, validar la gramática del mismo y llamar al generador pasándole la ruta de destino. El nombre del fichero generado es idéntico al de la conversión a USE, aunque con .uml como sufijo, y también tiene en cuenta lo de no sobrescribir los ficheros del mismo nombre.

En este apartado, además del problema que se comentará a continuación, no se ha encontrado complicaciones mayores que la necesidad de formarse en nuevos lenguajes y tecnologías.

6.2.1. Transformación del código OCL

En la tercera iteración del desarrollo se buscaba la integración de OCL en la herramienta. Esto no supuso ningún tipo de problema para el paso a USE, ya que UML2 trata el código OCL como un simple campo de texto. Sin embargo, Xtext no es capaz de detectarlo correctamente, ya que la gramática de USE no utiliza ningún tipo de separadores concretos para diferenciarlo. El tipo texto de Xtext tampoco era útil, ya que solo detectaba el texto incluido entre dos comillas simples.

Para solucionarlo, primeramente se pensó en obtener todo el texto, independientemente de lo que sea, hasta la aparición de cualquier otra regla de la gramática. Esto, después de una consulta exhaustiva en los foros, se ve que no es posible actualmente.

La siguiente idea fue la de utilizar el tipo texto de Xtext y procesar previamente el archivo para colocar comillas simples antes y después de cada sentencia. Esto, aunque parecía la solución más sencilla, conllevaba muchos problemas, ya que USE permite colocar todo su código en una misma línea si así lo desease el usuario y realmente existían muchos elementos capaces de ser el final de la sentencia. Finalmente se descartó este método tanto por su complejidad como por la poca seguridad que proporcionaba.

Después de esto se buscó ayuda externa publicando una pregunta en el foro de Eclipse [39], aunque no se llegó a encontrar una buena solución.

Posteriormente se descubrieron los plugin de Eclipse para OCL y, concretamente, se intentó utilizar el llamado *EssentialOCL*, que describe toda la gramática de OCL sin añadir florituras. En un primer momento se intentó heredar directamente el plugin, pero finalmente se descubrió que las referencias cruzadas no funcionaban correctamente al heredar gramáticas [40], lo que generaba muchísimos errores.

La solución definitiva fue encontrar el código fuente tanto del plugin *EssentialOCL* [41] como de otro plugin llamado *Base* [42], del que depende, y copiar en la gramática de USE todas las reglas necesarias para leer correctamente OCL. Además, como la herramienta no necesita validarlo, se han sustituido todas las referencias cruzadas por simples identificadores (es decir, cualquier palabra) para evitar problemas. Por último, en el generador, se ha vuelto a pasar todo el contenido de estas clases a texto, teniendo en cuenta todos los símbolos que no se almacenan en ellas.

Respecto al SOIL, como es un lenguaje que sí tiene unos separadores definidos (*begin-end*) se ha decidido establecer su código como texto y utilizar la idea de

las comillas simples antes mencionada que, en este caso, no acarrea apenas problemas.

6.3. Unificación de conversiones e interfaz

Para unificar ambas transformaciones se ha implementado un pequeño método que recibe dos parámetros de texto: la ruta al fichero fuente y la ruta en la que se almacenará el archivo generado. Después de comprobar que esta información es correcta (no es un texto nulo o vacío, las rutas son válidas y el fichero es .uml o .use) envía, dependiendo del sufijo, la información a uno u otro conversor.

Respecto a la interfaz (Figura 29), esta se ha diseñado lo más simple posible con el propósito de que el usuario entienda su funcionamiento de un único vistazo.

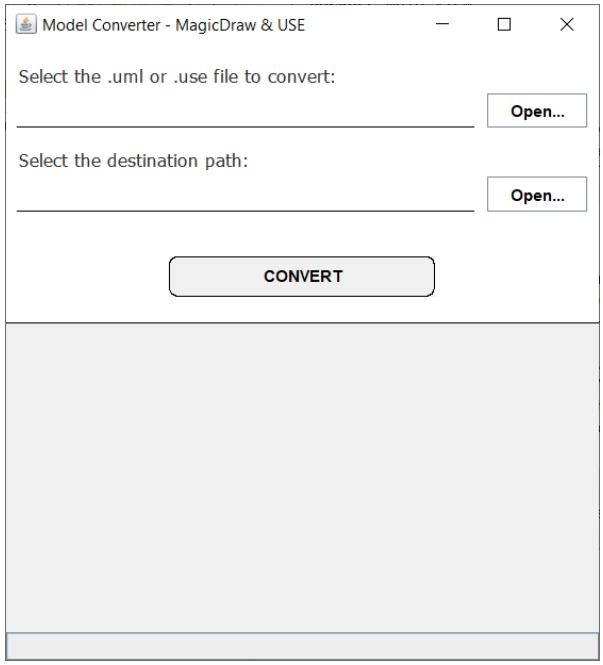


Figura 29. Interfaz de la herramienta.

Contiene únicamente dos campos de texto que se pueden rellenar tanto manualmente como mediante un selector de ficheros y carpetas (Figura 30). Además, este selector, que corresponde a un *JFileChooser* [43], está configurado para restringir la selección a ficheros en el primer caso y a carpetas en el segundo.

Como añadido, en la selección de ficheros se han añadido filtros para encontrar más fácilmente los archivos .uml y .use.

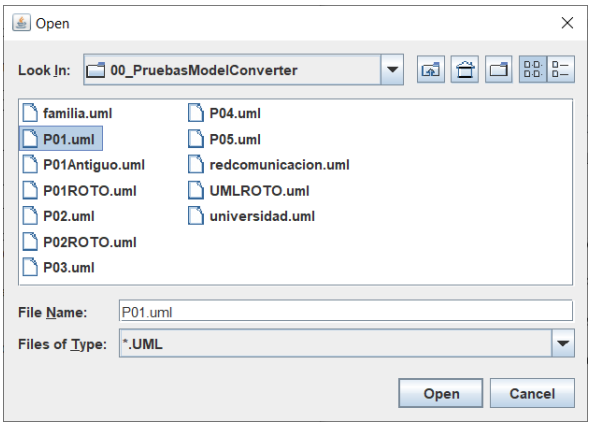


Figura 30. Selector de archivos/carpetas JFileChooser.

Con el objetivo de que el programa no diese la sensación de haberse quedado congelado mientras está convirtiendo el archivo, se ha añadido una barra de progreso en la parte inferior (Figura 31). Esta, debido a la duración incierta del programa al no trabajar directamente con hebras, no escribe un porcentaje concreto, sino que se mueve de un lado para otro. Aunque no es lo ideal, sí que cumple con la función de informar al usuario de que se está ejecutando su conversión y ahorra modificar enormemente la aplicación para trabajar íntegramente con hebras. En su lugar, únicamente se crea una hebra al pulsar el botón de *Convert*, activando la barra de progreso y deshabilitando el botón para evitar interferencias entre hebras.

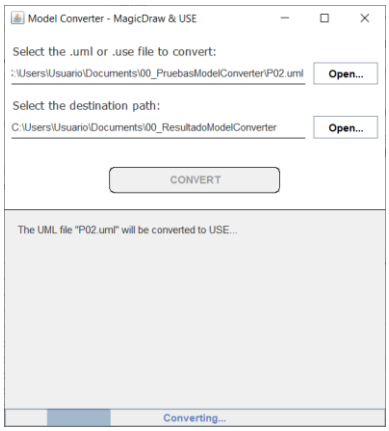


Figura 31. Interfaz de la herramienta en el proceso de conversión.

Otro punto a destacar de la interfaz es cómo muestra a tiempo real los errores y advertencias que se van encontrando durante la conversión, en lugar de ponerlos todos al terminar. Para ello lo habitual es el uso de hebras, pero como esto, al igual que con el punto anterior, suponía un gran cambio en el diseño de la aplicación, se ha optado por migrar la consola al área de texto que muestra los mensajes. De esta forma, cualquier texto impreso por consola se mostrará directamente en este área.

Por último, es importante aclarar que tras cada conversión la consola no se limpia. Esto está hecho a sabiendas, ya que, aunque a algunos les parecerá algo molesto, puede ser realmente útil para alguien que esté transformando varios archivos seguidos. Así, podría saber fácilmente que archivos ya ha convertido y podría consultar las advertencias al final sin que desaparezcan cada vez que pulse el botón. Se puede ver este funcionamiento en la Figura 32.

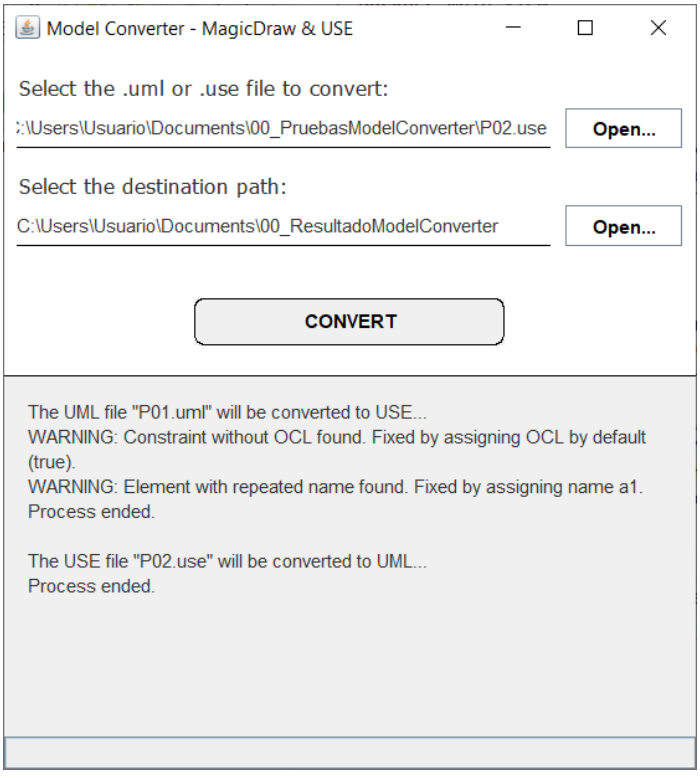


Figura 32. Interfaz de la herramienta tras finalizar dos conversiones.

7

Validación y pruebas

En este capítulo se relatará el proceso de validación y pruebas de la herramienta, explicando las distintas problemáticas que se han encontrado, las decisiones tomadas al respecto y las pruebas que finalmente se han realizado.

7.1. Algoritmo de equivalencia de modelos

El gran problema que se encontró a la hora de automatizar las pruebas del programa fue que dos archivos XMI o USE podían ser equivalentes aunque el texto que contuviesen no fuera igual. Esto es debido a que tanto USE como XMI permiten ordenar sus elementos de múltiples formas, siendo todas ellas válidas.

Primeramente se pensó en comprobar la igualdad entre ficheros del mismo tipo obteniendo los objetos EMF generados por Xtext y UML2 y comparándolos

directamente con el método *equals*. Esto no funcionó en ninguno de los dos casos. Los objetos EMF generados por Xtext devolvían siempre *true,* independientemente de cómo fuesen los modelos, y los generados por UML2 no se consideraban iguales por la discrepancia entre identificadores.

La solución a este problema fue crear un conjunto de clases que actuaran como un pivote auxiliar entre los ficheros. Este pivote, denominado *T*, solo almacena la información necesaria para comparar los modelos, utilizando únicamente campos de texto, booleanos, listas y, si es estrictamente necesario, referencias a otras clases del pivote. Dentro de cada una de estas clases se ha añadido un método *equals,* que compara dos objetos teniendo en cuenta todas las posibles diferencias que no implican la desigualdad de los modelos, como elementos sin nombre o restricciones sin OCL.

En la Figura 33 se muestra el modelo de clases de T sin incluir las operaciones, para un mayor entendimiento de la estructura del pivote por parte del lector.

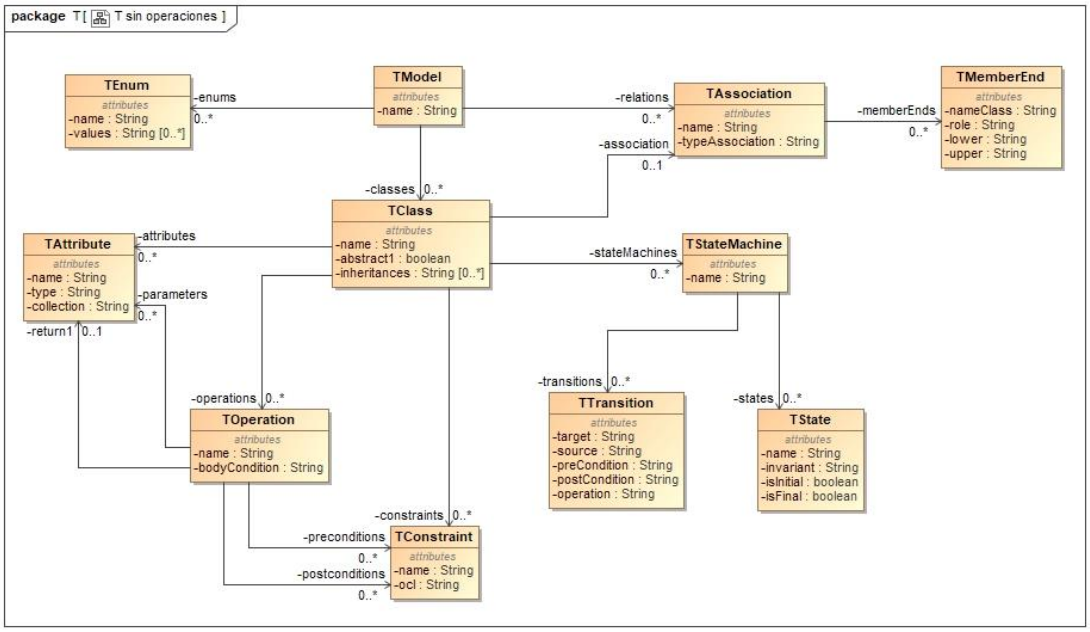


Figura 33. Modelo de clases del pivote T sin operaciones

Debido a sus dimensiones, en el apéndice D se puede ver el modelo completo, que incluye en cada clase las operaciones estrictamente necesarias: constructores, *setters* (establecen los atributos), el *equals*, el *hashcode* y, en algunos casos, métodos extra para devolver información.

El siguiente paso era almacenar la información procedente de cada fichero en el pivote T. Para ello se han creado dos métodos, uno para USE y otro para XMI, que, mediante su respectivo plugin, obtienen el objeto EMF del fichero recibido y lo recorren añadiendo toda la información de utilidad que contienen en un objeto *TModel*, sin realizar ningún tipo de modificación ni comprobación sobre el modelo.

Respecto al método de USE, hay que aclarar que el código OCL no se ha tenido en cuenta a la hora de almacenar la información, ya que supone recorrer un gran número de clases de bastante complejidad. En su lugar, para comprobar que la conversión de OCL funciona correctamente, en las pruebas unitarias se han realizado varias transformaciones sobre el mismo modelo y se han equiparado los resultados con el método que compara dos archivos XMI, que, como es un simple campo de texto, sí es capaz de comparar OCL.

Finalmente, se ha implementado una clase general que contiene tres métodos muy simples para comprobar la equivalencia entre modelos: uno recibe un fichero USE y un fichero XMI, otro dos ficheros XMI y el último dos ficheros USE. Estos métodos únicamente crean un objeto *TModel* para cada archivo, a través de uno de los métodos antes mencionados, y los compara utilizando el método *equals*.

De esta manera se ha obtenido un algoritmo capaz de comprobar la equivalencia tanto entre ficheros del mismo tipo como de diferente.

7.2. Desarrollo de pruebas

Para desarrollo de las pruebas se han tenido en cuenta las siguientes cuestiones.

En primer lugar, la herramienta recibe como dato de entrada ficheros XMI o USE que representan modelos. Estos archivos no pueden ser generados automáticamente, ya que supondría la implementación de un algoritmo bastante complejo que implicaría invertir un tiempo mucho mayor al beneficio que se obtendría.

Por otra parte, tampoco es posible crear un gran número de ficheros de prueba manualmente, tanto por el tiempo que conlleva como por la infinitud de las posibilidades.

Debido a todo esto, se han seleccionado razonadamente los casos de prueba que deben cumplir los ficheros creados, de tal forma que se pueda comprobar todo el funcionamiento con pocos ficheros.

7.2.1. Plan de pruebas

El plan de pruebas de la aplicación, que ha ido ampliándose en cada iteración hasta obtener el listado final, contiene todas las posibilidades que son necesarias comprobar para garantizar el buen funcionamiento de la herramienta.

En este caso, consiste en una tabla que contiene el nombre del caso de prueba, una pequeña descripción del mismo y si puede representarse directamente en cada gramática. En caso de no poder hacerlo especifica en su lugar su equivalencia. Por otra parte, la celda correspondiente al nombre se ha rellenado de un determinado color para indicar de forma visual el archivo en el que está incluido, los cuales se describirán en el próximo apartado.

Debido a su extensión, esta tabla ha sido anexada como apéndice F, pero se ruega encarecidamente al lector que la consulte antes de continuar.

7.2.2. Ficheros generados

A partir del plan de pruebas obtenido se fueron agrupando los distintos casos en ficheros, de forma que cada uno se centrase en comprobar ciertos elementos concretos.

En la siguiente tabla se exponen dichos ficheros, indicando su nombre, los elementos generales en los que se centra, los casos de prueba que cubra y una pequeña descripción del modelo. Para mayor detalle, estos pueden ser consultados dentro del código fuente de la herramienta, en una carpeta externa llamada *testFiles*.

N.	GENERAL	CASOS	DESCRIPCIÓN
P01	Atributos simples y asociación reflexiva	C2, A4-A20, A29-A32, R2, R3, R5, S2, S7, S12, S13, Z1	Clase abstracta con todos los atributos simples disponibles (sin colecciones) y una restricción dentro de la clase sin ocl y sin nombre. No hay operaciones ni máquinas de estados. Asociación reflexiva 1 : 0..1 sin roles. Habrá 3 atributos inicializados, uno con OCL, otro con un entero y otro con un real. También habrá un atributo derivado.
P02	Clases y atributos colección	C3, C4, C5, C9, C10, C11, C13, E3, A21-A28, S8, S19	Clase sin nombre que hereda de una clase abstracta sin nombre. Entre ellas hay una associationClass sin nombre y abstracta (sin multiplicidades explicitas y con roles). La associationClass tendrá un atributo de cada tipo de colección, algunos sin nombre y otros con nombre entremezclados.

P03	Enumerados, asociaciones y atributos de referencia	C1, C7, C8, C12, E1-E4, A1, A2, A3, S1-S6, S8-S18, S20, S21, Z2-Z4	Clase simple con un atributo de cada tipo de referencia. Además habrá un enumerado sin valores. otro sin nombre con un valor y otro con 5 valores. Habrá otra clase simple con la que la clase anterior tendrá todos los tipos de asociaciones, incluida la AssociationClass. Una AssociationClass heredará de otra. Algunas tendrán roles, otras ninguno y otras uno si y uno no. Se utilizarán todas las multiplicidades disponibles. Habrá nombres repetidos y vacíos entre las clases, las asociaciones, los enumerados y los roles (se añade una clase con nombre vacío para cumplir esto)
P04	Operaciones y restricciones	C1, C6, O1-O20, R1, R5, R6, R7	Clase con todas las posibilidades de operaciones posible y restricciones fuera y dentro de la clase con y sin ocl, algunas con nombre repetido. También restricciones en el contexto de operación. Otra clase heredará de esta.
P05	Máquinas de estado	M1-M22	Una clase con todas las posibilidades de máquinas de estado.
P06	OCL	Z5	Modelo complejo con mucho OCL, solo versión USE
P07	OCL	Z5	Modelo complejo con mucho OCL, solo versión USE
P08	OCL	Z5	Modelo complejo con mucho OCL, solo versión USE
P09	OCL	Z5	Modelo complejo con mucho OCL, solo versión USE
P10	Errores	I6, I7	Error gramatical en el fichero

Tabla 1. Ficheros generados para las pruebas.

7.2.3. Pruebas finales

Finalmente, disponiendo tanto del algoritmo de equivalencia como de los ficheros, se procedió a escribir las pruebas unitarias necesarias en JUnit5.

Primeramente, se configuró que, antes de que se ejecutase el conjunto de pruebas, se eliminaran todos los archivos existentes en la carpeta de destino y se

comprobase la equivalencia de las dos versiones de los archivos creados manualmente.

Seguidamente, se realizaron las pruebas de interfaz, correspondientes a los casos de prueba I1-I5, tomando como referencia el mensaje recibido al ejecutar el error. De igual forma, para los ficheros P10 se comprobó que no saltase ninguna excepción.

Para cada uno de los ficheros P01 a P05 se han creado cuatro pruebas. Las dos primeras transforman cada una de las versiones del modelo y luego comprueban si el fichero generado es equivalente tanto al archivo con el que ha sido creado como a la versión de su mismo tipo creada manualmente. Las otras dos pruebas vuelven a transforman estos dos archivos generados y los comparan con su archivo original.

Los ficheros P06 a P09 pretenden comprobar la correcta conversión del código OCL. Como el algoritmo solo es capaz de comparar OCL cuando equipara dos archivos XMI, se ha transformado cada archivo de USE a XMI, de XMI a USE y de USE otra vez a XMI. Si ambos archivos XMI son equivalentes significa que la herramienta es capaz de traspasar el OCL correctamente entre transformaciones.

Todas estas pruebas fueron ejecutadas hasta la desaparición de fallos, corrigiendo los errores pertinentes por el camino. Por último, de forma manual, se importaron todos los ficheros generados en sus respectivas herramientas, con el fin de asegurar definitivamente el correcto funcionamiento de la herramienta.

8

Conclusiones

Para finalizar esta memoria, se comentarán las conclusiones a las que se ha llegado durante el desarrollo de la herramienta y se listarán una serie de posibles líneas para ampliar la herramienta en un futuro.

8.1. Desarrollo de la herramienta

El desarrollo de este proyecto ha supuesto el aprendizaje de nuevas tecnologías y conceptos antes desconocidos por la autora, que lo eligió por su interés en la Ingeniería de Software Dirigida por Modelos y en las posibilidades que ofrece las capacidad de análisis de la herramienta USE.

En primer lugar, ha significado la familiarización con el uso de componentes en Eclipse y el descubrimiento de varios de ellos relacionados con el mundo del modelado de software, como es el caso de EMF y UML2. De igual forma, se ha aprendido mucho respecto a gramáticas y lenguajes, sobre todo con la formación

y el uso de Xtext, obteniendo conocimientos de primera mano respecto a la creación y aplicación de Lenguajes Específicos de Dominio.

Asimismo, aparte de Java, se ha utilizado Xtend, que ha resultado ser un lenguaje muy útil que se podría utilizar más frecuentemente para trabajar con Java.

El proceso de aprendizaje no ha sido tarea fácil, ya que algunos plugins no tenían una buena documentación o eran muy antiguos, lo que ha provocado búsquedas constantes en el foro de Eclipse y la necesidad de depuración de los objetos EMF, junto a la consulta del código de sus clases, para encontrar la información que se requería.

Aun así, sin lugar a dudas, la parte más costosa y que mayor esfuerzo y tiempo ha supuesto ha sido la integración del estándar OCL en la gramática de USE, ya que no se terminaba de encontrar una solución adecuada y fácil de implementar.

Por otro lado, también se ha requerido un análisis exhaustivo de ambas gramáticas, lo que ha ampliado enormemente el conocimiento que se tenía sobre MagicDraw y USE, sumado al descubrimiento y estudio del estándar de intercambio de modelos XMI.

Esto último, además, ha motivado la búsqueda de la equivalencia entre modelos a pesar de las diferencias entre gramáticas, corrigiendo también los posibles errores por parte del usuario con el propósito de obtener siempre modelos válidos.

Del mismo modo, en todas las fases e iteraciones del desarrollo se han aplicado los conocimientos obtenidos en la carrera, mejorándolos en algunos casos.

En definitiva, se está muy satisfecho con el resultado final de la herramienta y con la experiencia y conocimientos obtenidos durante su desarrollo.

8.2. Líneas futuras de ampliación

A pesar de su buen funcionamiento, la herramienta aún puede ser ampliada y mejorada en un futuro, de tal forma que abarque las limitaciones antes mencionadas y otros aspectos que resultarían de utilidad a la hora de intercambiar los modelos.

En primer lugar, sería conveniente añadir, a modo de plugin, dos opciones en la interfaz, tanto de MagicDraw como de USE, que permitan importar y exportar directamente en el formato contrario. Esto ahorraría tener que exportar el modelo y pasarlo manualmente por la herramienta.

En segundo lugar, se podrían reducir las limitaciones de la misma corrigiendo el problema de las relaciones n-arias de USE, que actualmente se consideran como binarias, y/o ampliando la conversión de las máquinas de estados con estados compuestos y de submáquinas. Para esto último sería necesario desarmar estos estados manteniendo su lógica, obteniendo máquinas que contengan únicamente estados simples y, por tanto, puedan ser representadas en lenguaje USE.

En tercer lugar, sería muy útil transformar también la colocación de los elementos en su respectivo diagrama o al menos generar uno donde se sitúen automáticamente de una forma coherente y agradable visualmente. Esto ahorraría tener que asentar los objetos manualmente en el diagrama.

Por último, la herramienta podría ampliarse a otros lenguajes textuales y visuales que no tengan la opción de exportar/importar el formato pivote. La transformación de cualquier gramática a Eclipse UML2 XMI, el pivote, implica tanto su validez en herramientas visuales como MagicDraw como la posibilidad de intercambiar el modelo con USE y con el resto de lenguajes que se añadan a la herramienta.

9

Referencias

[1] Object Management Group, "About the Unified Modeling Language Specification Version 2.5.1", *Omg.org*, 2017. [Online]. Available: https://www.omg.org/spec/UML/About-UML/. [Accessed: 16- May- 2021].

[2] "USE: UML-based Specification Environment", *SourceForge*. [Online]. Available: https://sourceforge.net/projects/useocl/. [Accessed: 16- Jun- 2021].

[3] "MagicDraw - CATIA - Dassault Systèmes", *3ds.com*. [Online]. Available: https://www.3ds.com/products-services/catia/products/no-magic/magicdraw/. [Accessed: 24- Jun- 2021].

[4] R. Jolak et al., "Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication", *Empirical Software Engineering*, vol. 25, no. 6, pp. 4427-4471, 2020. Available: 10.1007/s10664-020-09835-6.

[5] J. Cabot, “Text to UML and other "diagrams as code" tools - Fastest way to create your models”, *Modeling Languages*, 2020. [Online]. Available: https://modeling-languages.com/text-uml-tools-complete-list/. [Accessed: 16- Jun- 2021].

[6] S. Meliá, C. Cachero, J. Hermida and E. Aparicio, “Comparison of a textual versus a graphical notation for the maintainability of MDE domain models: an empirical pilot study”, *Software Quality Journal*, vol. 24, no. 3, pp. 709-735, 2015. Available: 10.1007/s11219-015-9299-x.

[7] Object Management Group, “About the Object Constraint Language Specification Version 2.4”, Omg.org, 2014. [Online]. Available: https://www.omg.org/spec/OCL/2.4/About-OCL/. [Accessed: 22- Apr- 2021].

[8] F. Büttner and M. Gogolla, "On OCL-based imperative languages", *Science of Computer Programming*, vol. 92, pp. 162-178, 2014. Available: 10.1016/j.scico.2013.10.003.

[9] I. García-Magariño García, *Un marco para la definición y transformación de modelos en los sistemas multiagentes*, Madrid, 2010. [Online]. Available: https://eprints.ucm.es/id/eprint/10639/1/T31885.pdf

[10] F. Durán Muñoz, J. Troya Castilla and A. Vallecillo Moreno, “Desarrollo de software dirigido por modelos”, 2012. [Online]. Available: https://www.exabyteinformatica.com/uoc/Informatica/Tecnicas_avanzadas_de_ingenieria_de_software/Tecnicas_avanzadas_de_ingenieria_de_software_(Modulo_1).pdf [Accessed: 23- Jun- 2021].

[11] Object Management Group, “OMG | Object Management Group”, *Omg.org*. [Online]. Available: https://www.omg.org/index.htm. [Accessed: 19- Jun- 2021].

[12] Object Management Group, “Model Driven Architecture (MDA) | Object Management Group”, Omg.org. [Online]. Available: https://www.omg.org/mda/. [Accessed: 22- Jun- 2021].

[13] “Arquitectura Dirigida por Modelos (MDA)”, *Ingeniería del Software UAH*, 2015.[Online].Available:https://ingenieriadelsoftwareuah2015.wordpress.com/2015/03/23/arquitectura-dirigida-por-modelos-mda/. [Accessed: 21- Jun- 2021].

[14] Object Management Group, “About the MOF Query/View/Transformation Specification Version 1.3”, *Omg.org*. [Online].Available: https://www.omg.org/spec/QVT/About-QVT/. [Accessed: 24- Jun- 2021].

[15] F. Tomassetti, “The complete guide to (external) Domain Specific Languages – Strumenta”, *Strumenta*. [Online]. Available: https://tomassetti.me/domain-specific-languages/. [Accessed: 22- Jun- 2021].

[16] Object Management Group, "MetaObject Facility | Object Management Group", *Omg.org*. [Online]. Available: https://www.omg.org/mof/. [Accessed: 23- Jun- 2021].

[17] World Wide Web Consortium, “Extensible Markup Language (XML)”, *W3.org*, 2016. [Online]. Available: https://www.w3.org/XML/. [Accessed: 23- Jun- 2021].

[18] Oracle, “¿Qué es Java?”, *Java.com*. [Online]. Available: https://www.java.com/es/about/whatis_java.jsp. [Accessed: 16- Jun- 2021].

[19] E. SEAS, “Conoce el lenguaje de programación Java | Blog SEAS”, *Blog de SEAS*,2019.[Online].Available:https://www.seas.es/blog/informatica/conoce-el-lenguaje-de-programacion-java/. [Accessed: 18- Jun- 2021].

[20] C. Guindon, "Eclipse desktop & web IDEs | The Eclipse Foundation", *Eclipse.org*, 2021. [Online]. Available: https://www.eclipse.org/ide/. [Accessed: 16- Jun- 2021].

[21] Eclipse Foundation, “Eclipse 2021-03 | Eclipse Packages”, *Eclipse.org*. [Online].Available:https://www.eclipse.org/downloads/packages/release/2021-03/r/eclipse-modeling-tools. [Accessed: 16- Jun- 2021].

[22] “MDT/UML2 – Eclipsepedia”, *Wiki.eclipse.org*. [Online]. Available: https://wiki.eclipse.org/MDT/UML2. [Accessed: 16- Jun- 2021].

[23] R. Gronback, “Eclipse Modeling Project | The Eclipse Foundation”, *Eclipse.org*. [Online]. Available: https://www.eclipse.org/modeling/emf/. [Accessed: 16- Jun- 2021].

[24] Eclipse Foundation , “Papyrus”, *Eclipse.org*. [Online]. Available: https://www.eclipse.org/papyrus/. [Accessed: 16- Jun- 2021].

[25] Eclipse Foundation, “OCL – Eclipsepedia”, *Wiki.eclipse.org*. [Online]. Available: https://wiki.eclipse.org/OCL. [Accessed: 17- Jun- 2021].

[26] S. Efftinge and M. Spoenemann, “Xtext - Language Engineering Made Easy!”, *Eclipse.org*. [Online]. Available: https://www.eclipse.org/Xtext/. [Accessed: 17- Jun- 2021].

[27] S. Efftinge and M. Spoenemann, “Xtend - Modernized Java”, *Eclipse.org*, 2021. [Online]. Available: https://www.eclipse.org/xtend/. [Accessed: 17- Jun- 2021].

[28] “SOIL - USE: UML-based Specification Environment”, *Useocl.sourceforge.net*.[Online].Available:http://useocl.sourceforge.net/w/index.php/SOIL. [Accessed: 17- Jun- 2021].

[29] "JUnit 5", Junit.org. [Online]. Available: https://junit.org/junit5/. [Accessed: 17- Jun- 2021].

[30] O. Moreno, "Pruebas unitarias: imprescindibles para programar - Oscar Moreno", *Oscar Moreno*, 2019. [Online]. Available: http://oscarmoreno.com/pruebas-unitarias/. [Accessed: 17- Jun- 2021].

[31] Oracle, "StringBuilder (Java Platform SE 7)", *Docs.oracle.com*. [Online]. Available:https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html. [Accessed: 22- Jun- 2021].

[32] "GitHub: Where the world builds software", *GitHub*. [Online]. Available: https://github.com/. [Accessed: 20- Jun- 2021].

[33] "Git", *Git-scm.com*. [Online]. Available: https://git-scm.com/. [Accessed: 20- Jun- 2021].

[34] N. Singh and B. Rohit, "Build metamodels with dynamic EMF - IBM," 2007. [Online]. Available: https://www.ibm.com/developerworks/library/os-eclipse-dynamicemf/. [Accessed: 12- Mar- 2021].

[35] Database Systems Group Bremen University, "USE - A UML based Specification Environment", Db.informatik.uni-bremen.de, 2007. [Online]. Available: http://www.db.informatik.uni-bremen.de/projects/use/use-documentation.pdf. [Accessed: 20- Apr- 2021].

[36] S. Efftinge and M. Spoenemann, "Xtext - 15 Minutes Tutorial", *Eclipse.org*. [Online].Available:https://www.eclipse.org/Xtext/documentation/102_domainmodelwalkthrough.html. [Accessed: 08- Apr- 2021].

[37] S. Efftinge and M. Spoenemann, "Xtext - 15 Minutes Tutorial – Extended", *Eclipse.org*.[Online].Available:https://www.eclipse.org/Xtext/documentation/103_domainmodelnextsteps.html. [Accessed: 08- Apr- 2021].

[38] L. Hamann, O. Hofrichter and M. Gogolla, “On Integrating Structure and Behavior Modeling with OCL”, *Db.informatik.uni-bremen.de*, 2012.[Online].Available:http://www.db.informatik.uni-bremen.de/publications/Hamann_2012_MODELS.pdf. [Accessed: 24- Jun- 2021].

[39] “Eclipse Community Forums: TMF (Xtext) » Integrate ocl in xtext grammar or grab all the characters until any language rule appears | The Eclipse Foundation”, *Eclipse.org*,2021.[Online].Available:https://www.eclipse.org/forums/index.php/m/1841936/. [Accessed: 03- Jun- 2021].

[40] “Eclipse Community Forums: TMF (Xtext) » Integrate ocl in xtext grammar or grab all the characters until any language rule appears | The Eclipse Foundation”, *Eclipse.org*,2021.[Online].Available:https://www.eclipse.org/forums/index.php/m/1841936/. [Accessed: 03- Jun- 2021].

[41] Willink Transformations and others, “org.eclipse.ocl.git – OCL”, *Git.eclipse.org*,2010.[Online].Available:https://git.eclipse.org/c/ocl/org.eclipse.ocl.git/tree/plugins/org.eclipse.ocl.xtext.essentialocl/src/org/eclipse/ocl/xtext/essentialocl/EssentialOCL.xtext. [Accessed: 20- May- 2021].

[42] Willink Transformations and others, "org.eclipse.ocl.git - OCL", *Git.eclipse.org*,2010.[Online].Available:https://git.eclipse.org/c/ocl/org.eclipse.ocl.git/tree/plugins/org.eclipse.ocl.xtext.base/src/org/eclipse/ocl/xtext/base/Base.xtext. [Accessed: 24- May- 2021].

[43] Oracle, "JFileChooser (Java Platform SE 7)", *Docs.oracle.com*. [Online]. Available:https://docs.oracle.com/javase/7/docs/api/javax/swing/JFileChooser.html. [Accessed: 10- Jun- 2021].

Apéndice A

Manual de instalación

Para el correcto funcionamiento de la herramienta es obligatorio tener instalado Java en el dispositivo en el que se ejecute, utilizando como mínimo la versión de la máquina virtual de Java **(JRE) 8 Actualización 281** o el Kit de Desarrollo de Java **(JDK) 16.x.x** (con una de las dos es suficiente). Estas han sido las versiones utilizadas durante el desarrollo y, aunque es posible, no se garantiza su correcto funcionamiento para versiones anteriores.

La última versión de JRE puede descargarse desde el enlace https://www.java.com/es/download/ y la última versión de JDK desde https://www.oracle.com/java/technologies/javase-downloads.html.

Puesto que la herramienta es un simple ejecutable de Java, no necesita ningún tipo de instalación especial. Solo será necesario descargar el archivo .jar de los archivos proporcionarlos o del repositorio GitHub (https://github.com/juliarobles/model-converter) y darle doble *click* para iniciar el programa.

En caso de querer cargar el código en Eclipse para modificarlo o ampliarlo, se pueden descargar los proyectos desde estos mismos sitios e importarlos utilizando *File > Import > General > Existing Projects into Workspace.*

Apéndice B

Manual de usuario

En el presente documento se explicará el funcionamiento de la aplicación, cómo se exportan e importan los modelos tanto en MagicDraw como en USE, las recomendaciones sugeridas para la correcta transformación de los modelos y las limitaciones de la herramienta.

Uso de la aplicación

Una vez abierto el programa, aparecerá la pantalla de la Figura 1Figura 34.

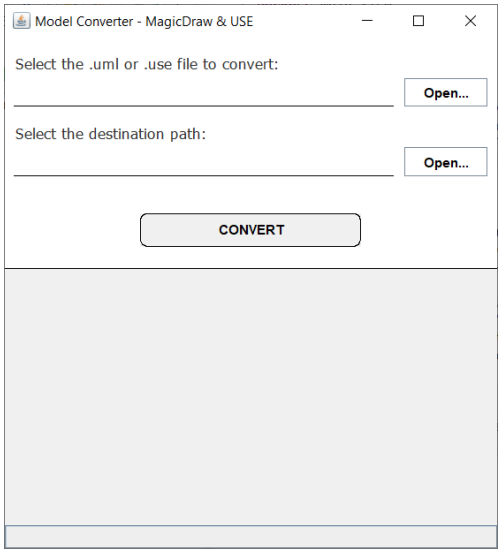


Figura 34. Interfaz de la herramienta.

El primer apartado indica la ruta del fichero que se quiere transformar. Esta puede introducirse tanto utilizando el botón de *Open*, que abrirá un selector de archivos, como de forma manual, copiando la ruta del explorador de archivos y pegándola en el campo de texto. En caso de utilizar el selector de archivos, hay

disponibles dos filtros, uno para .uml y otro para .use, que facilitarán la búsqueda de los archivos mostrando únicamente aquellos que terminen con el sufijo correspondiente.

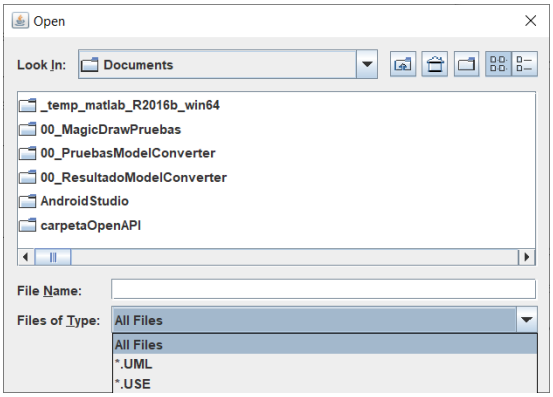


Figura 35. Filtros para la selección de ficheros/carpetas.

El segundo apartado indica la ruta de la carpeta en la que se almacenará el fichero generado por la herramienta. Esta puede introducirse tanto utilizando el botón de *Open*, que abrirá un selector de archivos, como de forma manual, copiando la ruta del explorador de archivos y pegándola en el campo de texto.

Después de rellenar correctamente ambos apartados se puede pulsar el botón de *Convert* para iniciar la conversión al lenguaje contrario. En el panel inferior aparecerá un mensaje que indicará el comienzo de la misma.

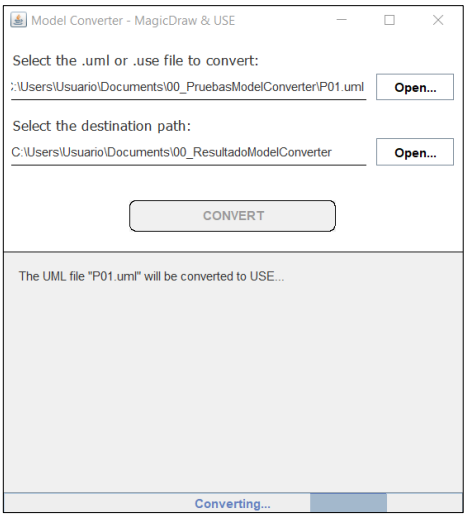


Figura 36. Interfaz al iniciar la conversión.

Si se encuentran advertencias o errores estas se mostrarán por pantalla. En caso de errores, no se generará ningún fichero, puesto que no se puede leer el recibido al no ser válido. Las advertencias, por su parte, únicamente avisan al usuario de los errores que se han encontrado en el modelo y como han sido corregidos, por si este quisiera arreglarlos en su modelo original.

Finalmente, se indicará la finalización de la conversión con el mensaje *Process ended.*

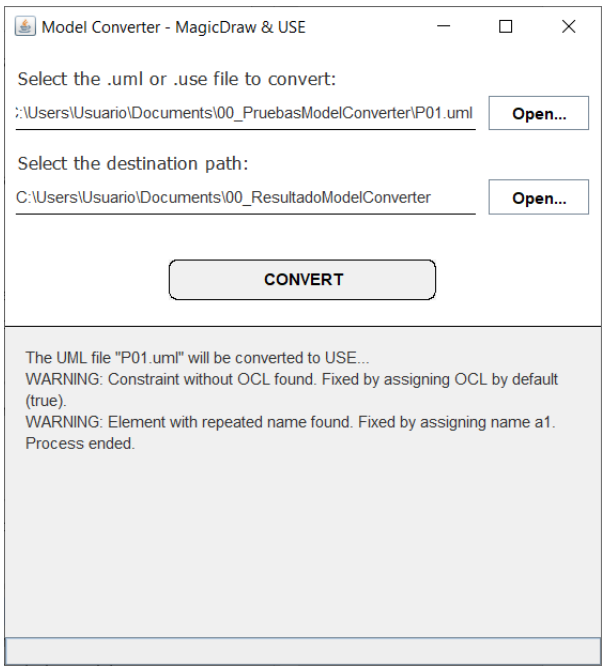


Figura 37. Interfaz al finalizar la conversión.

Seguidamente se puede repetir el proceso las veces que se deseen. El apartado informativo no se borrará hasta que se cierre la aplicación.

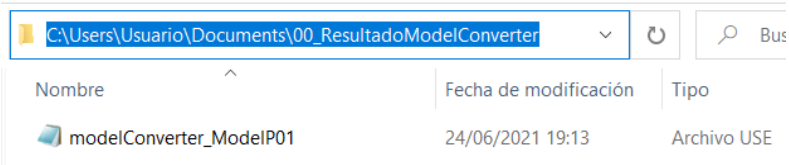


Figura 38. Archivo obtenido tras la conversión.

El archivo generado se puede encontrar en la carpeta seleccionada (Figura 38), con el nombre *modelConverter_X*, siendo X el nombre del modelo y no el del

archivo original. Si ya existía un fichero con este nombre, se habrá añadido un (N), donde N es el número de otros archivos con el mismo nombre. Por ejemplo, si se transforma un modelo USE que indica en su primera línea *model NombrePrueba* y que ya se ha convertido dos veces anteriormente, el nombre del fichero generado será *modelConverter_NombrePrueba(2).uml*.

Exportar/Importar en MagicDraw

Para poder utilizar la herramienta, desde MagicDraw se tiene que exportar e importar mediante la opción Eclipse UML2 XMI.

La opción de exportación se encuentra en *File* > *Export To* > *Eclipse UML2 XMI File* > *Eclipse UML2 (v5.x) XMI File*. Esta opción normalmente aparece oculta, pero se puede mostrar clicando sobre la flecha inferior del desplegable de *File*. Obviamente esta acción debe realizarse teniendo un proyecto UML abierto.

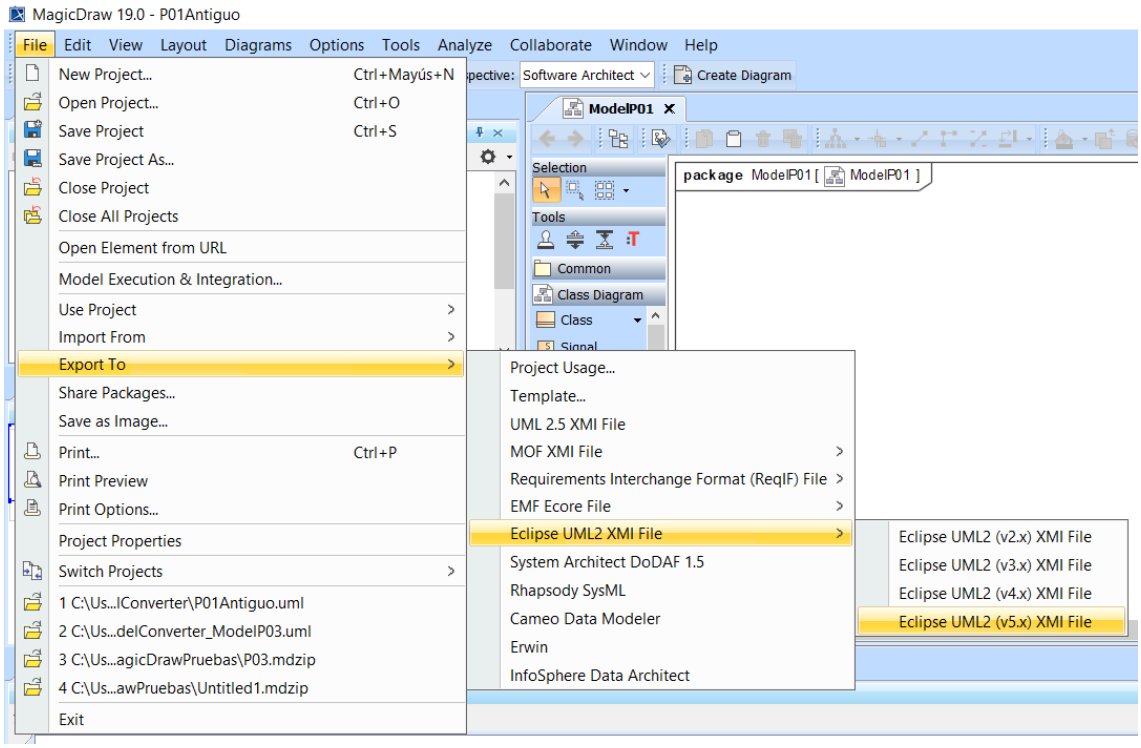


Figura 39. Exportar modelo desde MagicDraw.

Al pulsarla se generará en la carpeta indicada varios archivos, de los cuales únicamente nos interesará el que tenga el nombre de nuestro proyecto seguido del sufijo *uml*. Por ejemplo, de la Figura 40 solo necesitaríamos el fichero *Model.uml*. Este será el que tengamos que introducir en la herramienta.

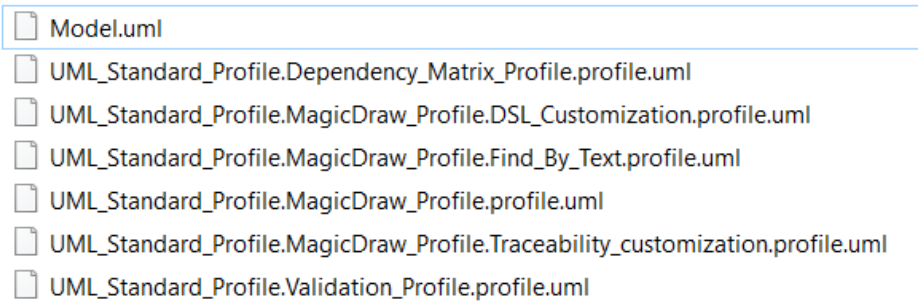


Figura 40. Archivos generados por MagicDraw.

Por otra parte, la opción de importación se encuentra en *File* > *Import From* > *Eclipse UML2 XMI File* > *Eclipse UML2 (v5.x) XMI File*. Tras pulsarlo se abrirá un selector de archivos donde se podrá seleccionar un fichero. En este caso se seleccionaría el fichero *uml* obtenido de la conversión de un modelo USE.

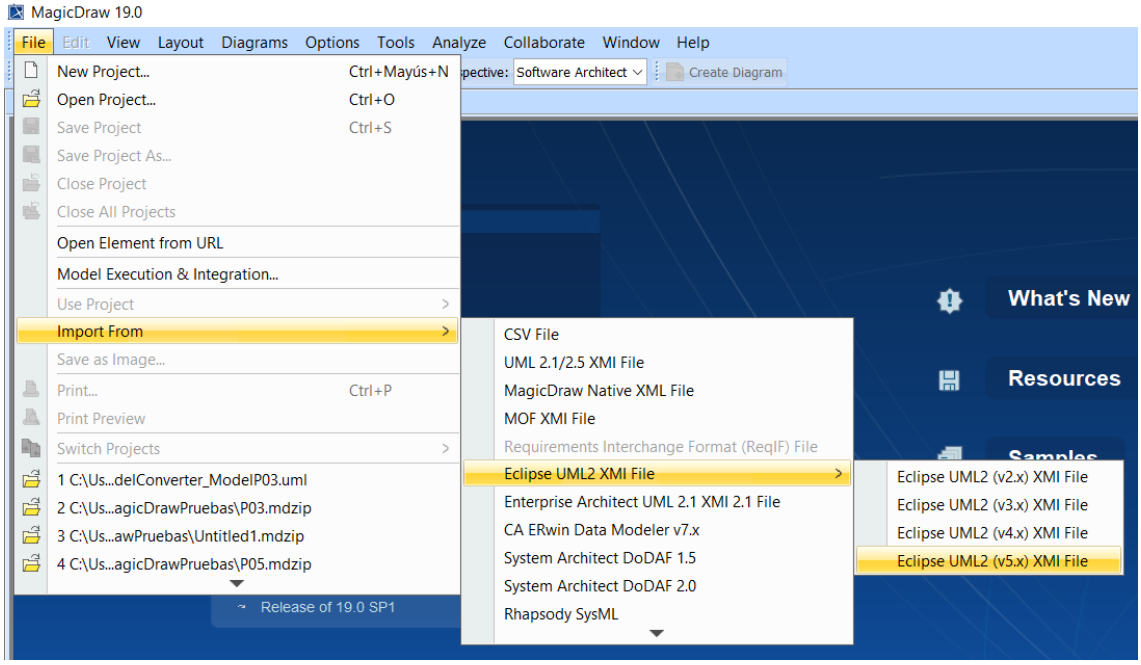


Figura 41. Importar archivo en MagicDraw.

Tras abrir se cargarán todos los elementos del modelo, aunque no se colocarán directamente en el diagrama. A partir de aquí el usuario debe únicamente diseñar el diagrama a su gusto, teniendo ya todos los elementos completos.

Una forma fácil de hacerlo es, tras crear el objeto diagrama mediante el botón *Create Diagram*, seleccionar todos los elementos excepto las relaciones y arrastrarlos hasta el diagrama para, posteriormente, seleccionarlos todos en el mismo y clicar en botón derecho > *Display* > *Display All Paths*. Esto colocará todas las relaciones existentes entre las clases. Si el resultado no fuese visualmente agradable siempre se puede modificar o colocarlo uno por uno desde el árbol de contenido.

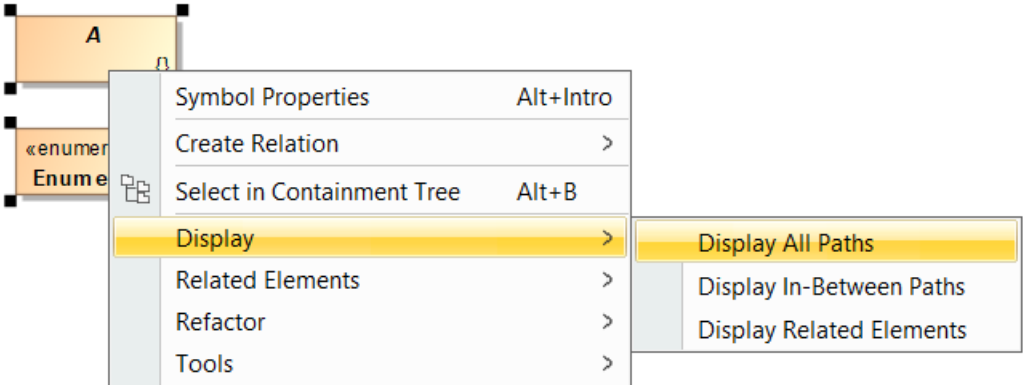


Figura 42. Colocación de las relaciones existentes en el diagrama.

Exportar/Importar en USE

La exportación e importación en la herramienta USE es mucho más sencilla, ya que realmente es el mismo proceso que cuando utilizamos USE de forma habitual. El archivo que se requiere para la conversión es el fichero *.use* que cargamos en la herramienta cuando queremos utilizar el modelo.

De igual forma, la herramienta genera un archivo *.use* con las mismas características, que también puede ser cargado por la herramienta.

Para su utilización, debemos clicar en el botón marcado en azul en la Figura 43 y abrir el archivo que contiene el modelo. Tras su correcta compilación se podrá utilizar toda la funcionalidad ofrecida por USE, como por ejemplo la creación de un diagrama visual mediante el botón marcado en verde en la Figura 43.

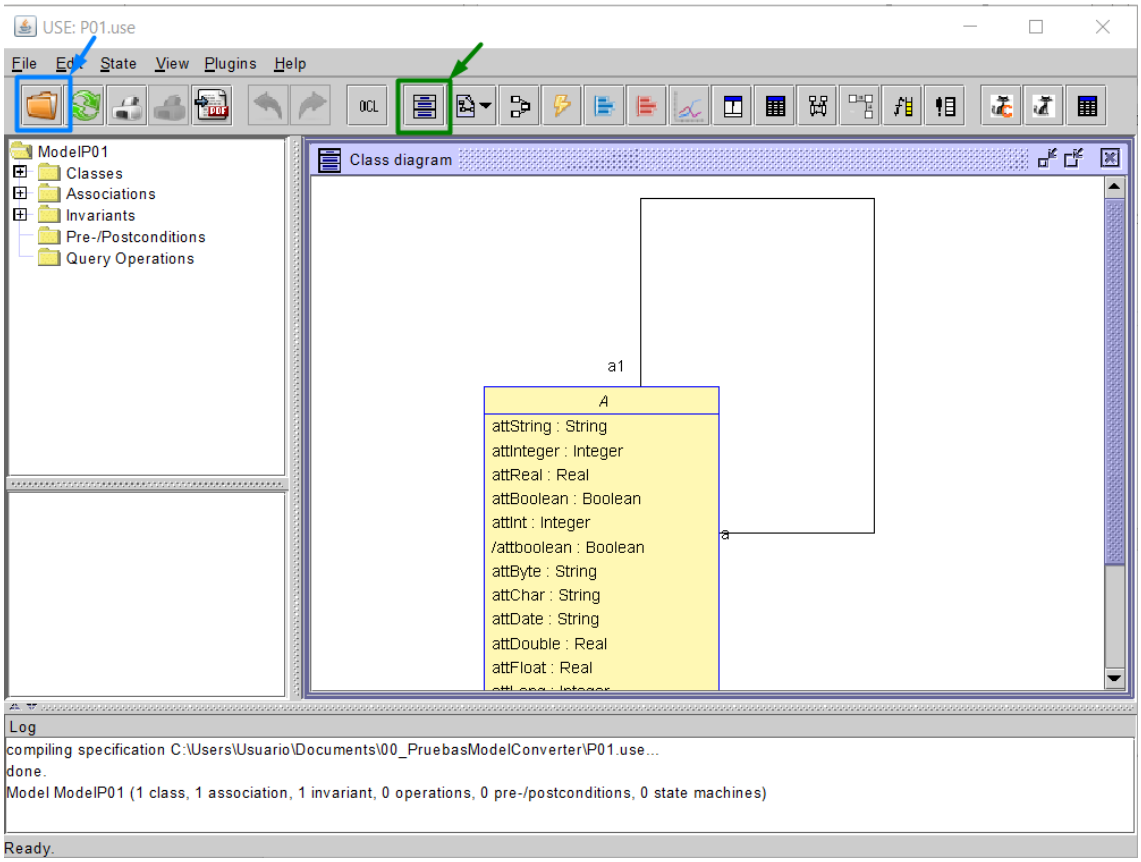


Figura 43. Carga de un modelo en la herramienta USE.

Recomendaciones para garantizar la conversión desde MagicDraw

En los modelos procedentes de MagicDraw se deben seguir ciertas pautas para asegurar su correcto intercambio a USE, ya que si no se introduce la información en los campos correspondientes esta no se tendrá en cuenta.

Primeramente, se debe evitar a toda costa repetir nombres de elementos o dejarlos vacíos. De igual forma, no se deben dejar atributos sin tipo y enumerados sin valores. Todo esto se comprueba y modifica en la aplicación, por lo que realmente

no debería haber ningún problema, pero siempre es conveniente tener el modelo lo más completo posible y evitar este tipo de fallos.

Por otro lado, si se quieren crear atributos o parámetros que contengan conjuntos, secuencias o bolsas en MagicDraw en primer lugar se debe establecer la multiplicidad máxima del atributo a cualquiera mayor que 1. Además, para una secuencia tendremos que marcar el campo *Is Ordered* mientras que para un conjunto será el campo *Is Unique*. En caso de que ambos campos sean falsos será una bolsa y, si ambos son verdaderos, la preferencia la tendrá la secuencia.

Si se quisiera añadir código SOIL u OCL en las operaciones, este debe ser introducido en el campo *Body Condition*. De igual forma, para inicializar atributos se debe usar el campo *Default Value* y, si el atributo es derivado, la expresión se añadirá al mismo campo y se activará el llamado *Is Derived*. Asimismo, todas las restricciones deben contener su expresión OCL en el campo *Specification*.

Para más información sobre los campos utilizados en MagicDraw se puede consultar el análisis de modelos desarrollado en la documentación de la herramienta.

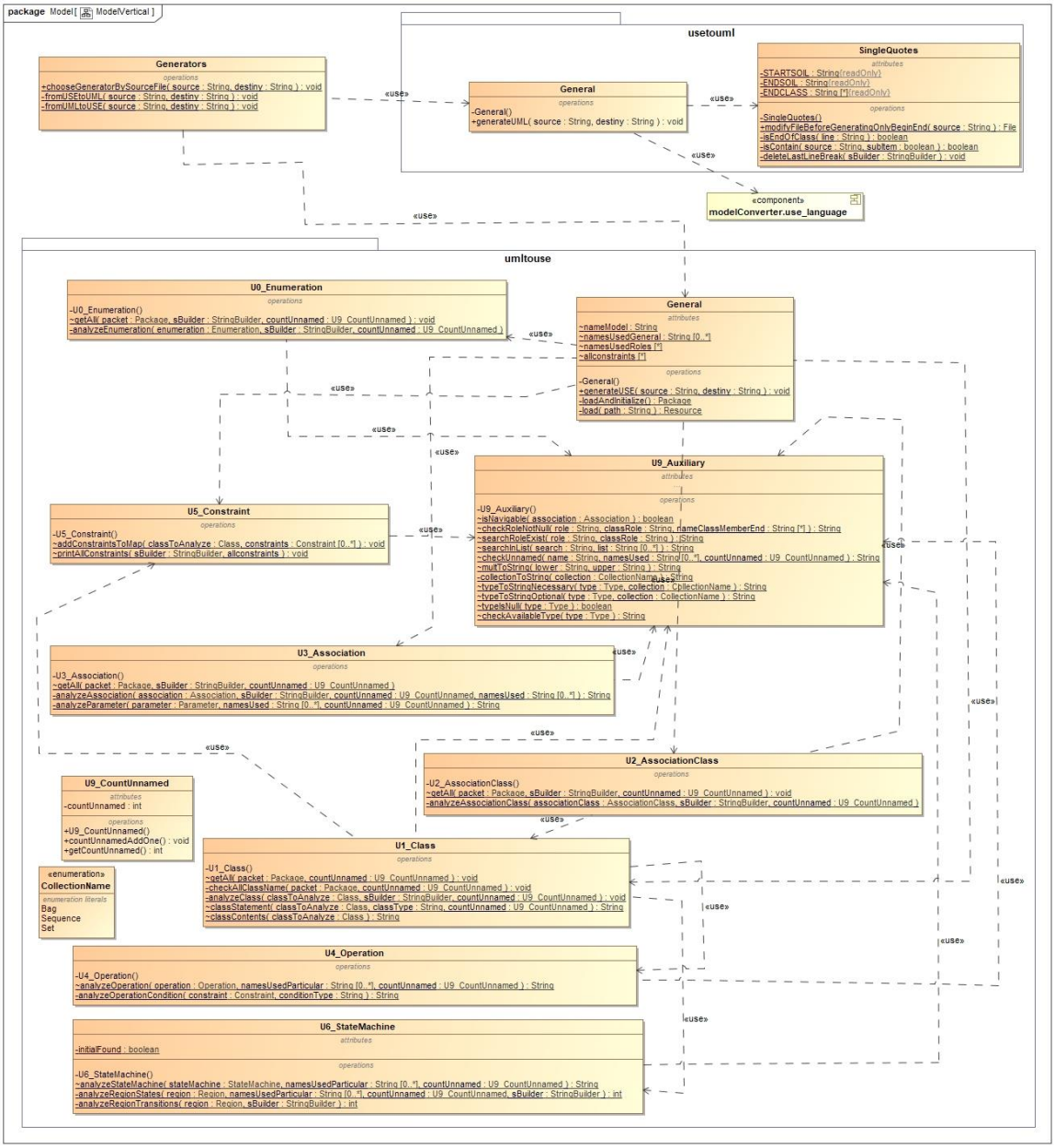
Limitaciones de la herramienta

Por la parte de USE, esta herramienta no es capaz de transformar relaciones N-arias, aunque sí puede hacerlo desde MagicDraw. De igual forma, tampoco se pueden convertir colecciones con más de un tipo.

Por la parte de MagicDraw, mucha información, como la visibilidad, no puede ser transformada a USE puesto que el lenguaje no lo permite. Asimismo, tampoco se pueden convertir los estados compuestos o de submáquinas.

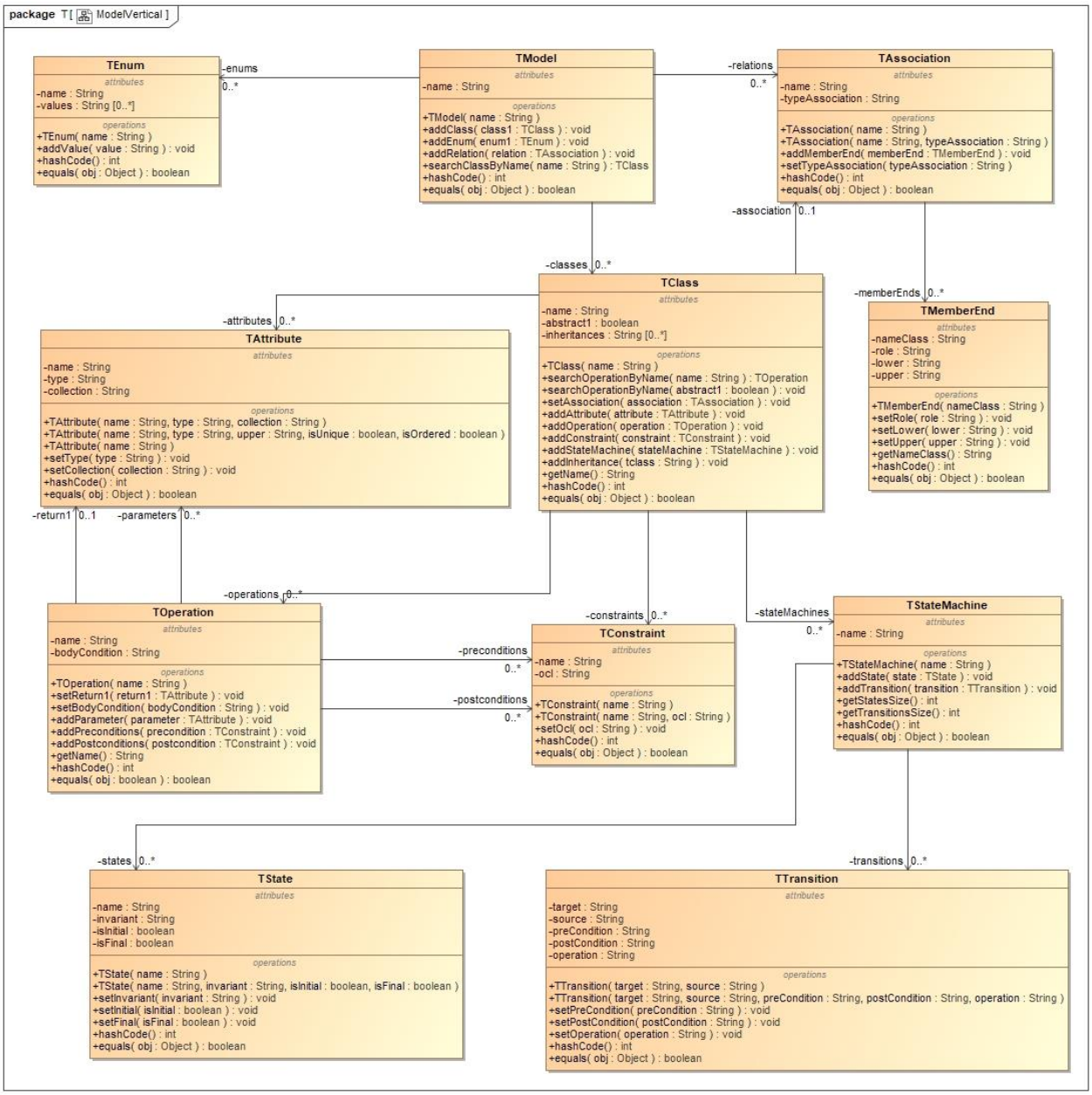
Apéndice C

Diagrama de clases modelo completo



Apéndice D

Diagrama de clases pivote T completo



Apéndice E

Requisitos finales

Requisitos funcionales

Alta prioridad

RF01. El usuario podrá intercambiar modelos que contengan únicamente el nombre del mismo en ambos sentidos.

RF02. El usuario podrá intercambiar modelos que contengan clases en ambos sentidos.

RF02.01. Las clase podrá tener un nombre.

RF02.02. Las clase podrá ser abstractas.

RF02.03. Las clase podrá heredar de una o más clases.

RF03. El usuario podrá intercambiar modelos que contengan atributos en ambos sentidos.

RF03.01. El atributo podrá tener un nombre.

RF03.02. El atributo podrá ser de tipo primitivo.

RF03.03. El atributo podrá ser de referencia a todo tipo de clases y enumerados.

RF03.04. El atributo podrá ser inicializados.

RF03.05. El atributo podrá ser derivados.

RF03.06. El atributo podrá tener multiplicidad mayor que uno.

RF03.07. El atributo podrá contener elementos ordenados, si su multiplicidad es mayor que uno.

RF03.08. El atributo podrá contener elementos únicos, si su multiplicidad es mayor que uno.

RF03.09. El sistema controlará que no se generen atributos sin tipo en el paso a USE.

RF04. El usuario podrá intercambiar modelos que contengan enumerados en ambos sentidos.

RF04.01. El enumerado podrá tener un nombre.

RF04.02. El enumerado podrá contener más de un valor.

RF05. El usuario podrá intercambiar modelos que contengan operaciones en ambos sentidos.

RF05.01. La operación podrá tener un nombre.

RF05.02. La operación podrá ser de tipo consulta.

RF05.03. La operación podrá tener un cuerpo.

RF05.03.01. El cuerpo será en código OCL para las operaciones de tipo consulta.

RF05.03.02. El cuerpo será en código SOIL para el resto de operaciones.

RF05.04. La operación podrá tener precondiciones.

RF05.04.01. La precondición podrá tener un nombre.

RF05.04.02. La precondición podrá tener una expresión OCL.

RF05.05. La operación podrá tener postcondiciones.

RF05.05.01. La postcondición podrá tener un nombre.

RF05.05.02. La postcondición podrá tener una expresión OCL.

RF05.06. La operación podrá devolver un tipo.

RF05.06.01. El tipo devuelto podrá ser de tipo primitivo.

RF05.06.02. El tipo devuelto podrá ser de referencia a todo tipo de clases y enumerados.

RF05.06.03. El tipo devuelto podrá tener de multiplicidad mayor que uno.

RF05.06.04. El tipo devuelto podrá contener elementos ordenados, si su multiplicidad es mayor que uno.

RF05.06.05. El tipo devuelto podrá contener elementos únicos, si su multiplicidad es mayor que uno.

RF05.07. La operación podrá tener uno o más parámetros.

RF05.07.01. El parámetro podrá tener un nombre.

RF05.07.02. El parámetro podrá ser de tipo primitivo.

RF05.07.03. El parámetro podrá ser de referencia a todo tipo de clases y enumerados.

RF05.07.04. El parámetro podrá ser inicializados.

RF05.07.05. El parámetro podrá ser derivados.

RF05.07.06. El parámetro podrá tener de multiplicidad mayor que uno.

RF05.07.07. El parámetro podrá contener elementos ordenados, si su multiplicidad es mayor que uno.

RF05.07.08. El parámetro podrá contener elementos únicos, si su multiplicidad es mayor que uno.

RF05.07.09. El sistema controlará que no se generen parámetros sin tipo en el paso a USE.

RF06. El usuario podrá intercambiar modelos que contengan restricciones en ambos sentidos.

RF06.01. La restricción podrá tener un nombre.

RF07. El usuario podrá intercambiar modelos que contengan asociaciones en ambos sentidos.

RF07.01. La asociación podrá tener un nombre.

RF07.02. La asociación podrá ser de tipo simple.

RF07.03. La asociación podrá ser de tipo agregación.

RF07.04. La asociación podrá ser de tipo composición.

RF07.05. La asociación podrá ser reflexiva.

RF07.06. La asociación podrá ser no navegable por un extremo.

RF07.07. La asociación podrá tener finales de asociación.

RF07.07.01. El final de asociación podrá referenciar a una clase.

RF07.07.02. El final de asociación podrá tener multiplicidad máxima.

RF07.07.03. El final de asociación podrá tener multiplicidad máxima.

RF07.07.04. El final de asociación podrá ser ordenado si la multiplicidad máxima es mayor que 1.

RF07.07.05. El final de asociación podrá tener un rol.

RF08. El usuario podrá intercambiar modelos que contengan clases de asociación en ambos sentidos.

RF08.01. La clase de asociación podrá tener un nombre.

RF08.02. La clase de asociación podrá ser abstracta.

RF08.03. La clase de asociación podrá heredar de una o más clases.

RF08.04. La clase de asociación podrá ser reflexiva.

RF08.05. La clase de asociación podrá tener finales de asociación.

RF08.05.01. El final de asociación podrá referenciar a una clase.

RF08.05.02. El final de asociación podrá tener multiplicidad máxima.

RF08.05.03. El final de asociación podrá tener multiplicidad máxima.

RF08.05.04. El final de asociación podrá ser ordenado si la multiplicidad máxima es mayor que 1.

RF08.05.05. El final de asociación podrá tener un rol.

Prioridad media-alta

RF08. El usuario podrá intercambiar modelos que contengan código OCL en ambos sentidos.

RF08.01. Una restricción podrá tener una expresión OCL.

RF08.02. Una postcondición podrá tener una expresión OCL.

RF08.03. Una precondición podrá tener una expresión OCL.

RF09. El sistema controlará en el paso a USE que no se generen elementos sin nombre, exceptuando las invariantes, precondiciones y postcondiciones.

RF10. El sistema controlará en el paso a USE que no se generen enumerados, clases o asociaciones con nombres repetidos respecto a otros enumerados, clases o asociaciones.

RF11. El sistema controlará en el paso a USE que no se generen roles repetidos respecto al resto de miembros de asociación que recibe como atributos la clase contraria.

RF12. El sistema controlará en el paso a USE que no se generen dentro del mismo contexto elementos con nombres repetidos.

RF13. El sistema controlará en el paso a USE que no se generen enumerados sin valores.

RF14. El sistema controlará en el paso a USE que no se generen tipos de atributos vacíos o nulos.

RF15. El sistema controlará en el paso a USE que no se generen tipos de parámetros vacíos o nulos.

RF16. El sistema controlará en el paso a USE que no se generen restricciones sin código OCL.

Prioridad media-baja

RF17. El usuario podrá seleccionar el fichero que se transformará a la gramática contraria independientemente seleccionando su ruta.

RF18. El usuario podrá seleccionar la carpeta de su ordenador donde se almacenará el fichero generado.

RF19. El sistema comprobará que el campo que contiene el fichero fuente no es vacío o nulo.

RF20. El sistema comprobará que el campo que contiene la ruta de la carpeta destino no es vacía o nula.

RF21. El sistema comprobará que el campo que contiene el fichero fuente almacena una ruta válida en el ordenador del usuario.

RF22. El sistema comprobará que el campo que contiene la ruta de la carpeta destino almacena una ruta válida en el ordenador del usuario.

RF23. El sistema comprobará que el campo que contiene el fichero fuente almacena la ruta a un fichero con un sufijo válido.

RF23.01. El sufijo podrá ser .uml.

RF23.02. El sufijo podrá ser .use.

RF24. El sistema alertará al usuario de los errores gramaticales que se encuentren en el fichero.

RF25. El sistema mostrará advertencias cuando, en las comprobaciones, modifique elementos en el modelo.

Prioridad baja

RF26. El usuario podrá intercambiar modelos que contengan máquinas de estados en ambos sentidos.

RF26.01. La máquina de estados podrá tener un nombre.

RF26.02. La máquina de estados podrá contener estados simples.

RF26.02.01. El estado podrá tener un nombre.

RF26.02.02. El estado podrá ser inicial.

RF26.02.03. El estado podrá ser final.

RF26.02.04. El estado podrá ser simple.

RF26.02.05. El estado podrá ser restringido por una expresión OCL, llamada invariante.

RF26.03. La máquina de estados podrá contener transiciones.

RF26.03.01. La transición tendrá un único estado fuente.

RF26.03.02. La transición tendrá un único estado objetivo.

RF26.03.03. La transición podrá referenciar la operación que la dispara.

RF26.03.04. La transición podrá tener una expresión OCL que funcione como precondición.

RF26.03.05. La transición podrá tener una expresión OCL que funcione como postcondición.

RF26.04. El sistema comprobará que la máquina de estados no tenga estados con nombre vacío o nulo.

RF26.05. El sistema comprobará que la máquina de estados no tenga estados con nombre repetido.

RF26.06. El sistema comprobará en el paso a USE que la máquina de estados tenga como mínimo dos estados y una transición.

Requisitos no funcionales

RNF01. Los ficheros con sufijo .uml procederán de la herramienta MagicDraw.

RNF02. Los ficheros con sufijo .use procederán de la herramienta USE.

RNF03. El sistema utilizará una gramática pivote para la transformación en ambos sentidos.

RNF04. El sistema se desarrollará en Java.

RNF05. El sistema solo podrá ser ejecutado sobre la máquina virtual de Java.

RNF06. La interfaz estará en inglés.

RNF07. La interfaz será lo más simple posible.

RNF08. El sistema será fácil de usar.

RNF09. El sistema será de fácil aprendizaje.

RNF10. El sistema se exportará en un JAR ejecutable.

Apéndice F

Plan de pruebas

N.	DESCRIPCIÓN	USE	MAGICDRAW
CLASES			
C1	Clase simple	SI	SI
C2	Clase abstracta	SI	SI
C3	Clase sin nombre	unnamed1	SI
C4	Clase abstracta sin nombre	unnamed1	SI
C5	Clase hereda de 1 clase	SI	SI
C6	Clase hereda de +1 clase (al menos una debe tener atributos, operaciones e invariantes)	SI	SI
C7	AssociationClass simple	SI	SI
C8	AssociationClass en relación reflexiva	SI	SI
C9	AssociationClass abstracta	SI	SI
C10	AssociationClass sin nombre	unnamed1	SI
C11	AssociationClass abstracta sin nombre	unnamed1	SI
C12	AssociationClass hereda de 1 clase	SI	SI
C13	+1 clase sin nombre, al menos una abstracta, al menos una AssociationClass	unnamedX	SI
ENUMERADOS			
E1	Enumerado sin nombre	unnamed1	SI
E2	Enumerado sin valores	requiredValue	SI
E3	Enumerado con 1 valor	SI	SI
E4	Enumerado con +1 valor	SI	SI

ATRIBUTOS			
A1	Atributo Class	SI	SI
A2	Atributo AssociationClass	SI	SI
A3	Atributo Enumeration	SI	SI
A4	Atributo String	SI	SI
A5	Atributo Real	SI	SI
A6	Atributo Boolean	SI	SI
A7	Atributo Integer	SI	SI
A8	Atributo int	Integer	SI
A9	Atributo boolean	Boolean	SI
A10	Atributo byte	String	SI
A11	Atributo char	String	SI
A12	Atributo date	String	SI
A13	Atributo double	Real	SI
A14	Atributo float	Real	SI
A15	Atributo long	Integer	SI
A16	Atributo short	Integer	SI
A17	Atributo StructuredExpression	String	SI
A18	Atributo UnlimitedNatural	String	SI
A19	Atributo void	String	SI
A20	Atributo vacio	String	SI
A21	Atributo sin nombre	unnamed1	SI
A22	+1 atributo sin nombre, otros con nombre por medio	unnamedX	SI
A23	Atributo Sequence(AllClassAndEnum)	SI	tipo X con return con multiplicidad > 1 y isOrdered y !isUnique
A24	Atributo Set(AllClassAndEnum)	SI	tipo X con return con multiplicidad > 1 y !isOrdered y isUnique
A25	Atributo Bag(AllClassAndEnum)	SI	tipo X con return con multiplicidad > 1 y !isOrdered y !isUnique
A26	Atributo Sequence(String)	SI	tipo String con return con multiplicidad > 1 y isOrdered y isUnique

A27	Atributo Set(Integer)	SI	tipo Integer con return con multiplicidad > 1 y !isOrdered y isUnique
A28	Atributo Bag(Real)	SI	tipo Boolean con return con multiplicidad > 1 y !isOrdered y !isUnique
A29	Atributo inicializado con un número entero	SI	SI
A30	Atributo inicializado con un número real	SI	SI
A31	Atributo inicializado con ocl	SI	SI
A32	Atributo derivado de ocl	SI	SI
OPERACIONES			
O1	Operación normal	SI	SI
O2	Operación normal sin nombre	unnamed1	SI
O3	Operación query	SI (solo con OCL y return)	isQuery = true
O4	Operación query sin nombre	unnamed1 (con OCL y return)	isQuery = true
O5	Operación normal con SOIL	SI	SI
O6	Operación query con OCL	SI	SI
O7	Operación con return tipo Simple	SI	SI
O8	Operación con return Class	SI	SI
O9	Operación con return AssociationClass	SI	SI
O10	Operación con return Enumerado	SI	SI
O11	Operación con 1 parámetro Simple	SI	SI
O12	Operación con +1 parámetro, simples, enumerado, class y associationClass	SI	SI
O13	Operación con precondición con ocl	SI	SI

O14	Operación con precondición sin nombre	SI	SI
O15	Operación con precondiciones nombres repetidos	NOMBRE1	SI
O16	Operación con precondición sin ocl	true	SI
O17	Operación con postcondición con ocl	SI	SI
O18	Operación con postcondición sin nombre	SI	SI
O19	Operación con postcondiciones nombres repetidos	NOMBRE1	SI
O20	Operación con postcondición sin ocl	true	SI
ASOCIACIONES			
S1	Asociación simple	SI	SI
S2	Asociación reflexiva	SI	SI
S3	Composición	SI	SI
S4	Agregación	SI	SI
S5	Asociación con un lado no navegable	Se convierte en atributo	SI
S6	Al menos 1 asociación cada tipo	SI	SI
S7	Asociación reflexiva sin roles	Uno es NOMBRECLASE 1	SI
S8	Asociación sin roles	SI	SI
S9	Asociación con roles	SI	SI
S10	Asociación uno con rol y otro no	SI	SI
S11	Multiplicidad 0	SI	SI
S12	Multiplicidad 1	SI	SI
S13	Multiplicidad [0..1]	SI	SI
S14	Multiplicidad [X..Y]	SI	SI
S15	Multiplicidad [0..*]	SI	SI
S16	Multiplicidad [1..*]	SI	SI
S17	Multiplicidad [*]	SI	SI

S18	Multiplicidad [X]	SI	SI
S19	Sin multiplicidades explicita	1	SI
S20	Final asociación ordenado	SI	SI
S21	Final AssociationClass ordenado	SI	SI
RESTRICCIONES:			
R1	Restricción con ocl	SI	SI
R2	Restricción sin ocl	true	SI
R3	Restricción sin nombre	SI	SI
R4	Restricción declarada dentro de la clase	SI	SI
R5	Restricción declarada fuera de la clase	SI	Solo se puede dentro
R6	Restricción en el contexto de una operación		
R7	Varias restricciones dentro y fuera de varias clases	SI	SI, aunque las de fuera estarán dentro
MAQUINAS DE ESTADOS			
M1	Solo estado inicial y final	SI	SI
M2	Inicial y mínimo un estado más, sin final	SI	SI
M3	Inicial, final y al menos dos estados más	SI	SI
M4	Maquina sin nombre	unnamed1	SI
M5	Más de una máquina sin nombre	unnamedX	SI
M6	Sin estados	NO (se comentará)	SI
M7	Solo 1 estado	NO (se comentará)	SI
M8	2 o más estados	SI	SI
M9	Estado sin nombre	unnamed1	SI
M10	Estado sin invariante	SI	SI
M11	Estado con invariante	SI	SI
M12	Más de un estado sin nombre	unnamedX	SI
M13	Estados con nombre repetidos	NOMBRE1	SI (aunque solo si uno es State y otro PseudoState)
M14	Sin transiciones	NO (se comentará)	SI
M15	Solo 1 transición	SI	SI

M16	Más de 1 transición	SI	SI
M17	Transición reflexiva	SI	SI
M18	Transición sin operación	SI	SI
M19	Transición con operación sin condiciones	SI	SI
M20	Transición con operación solo con precondición	SI	SI
M21	Transición con operación solo con postcondición	SI	SI
M22	Transición con operación con precondición y postcondición	SI	SI
OTRAS PRUEBAS SOBRE MODELOS			
Z1	Nombre con 1 carácter	SI	SI
Z2	Nombre con +100 caracteres	SI	SI
Z3	Nombres repetidos en enumerados, clases y asociaciones	NOMBREX	SI
Z4	Combinación de enumerados, clases y asociaciones sin nombre	unnamedX	SI
Z5	Correcto funcionamiento de OCL	SI	SI
INTERFAZ Y EXCEPCIONES			
I1	Error cuando campo archivo fuente vacío	Indiferente	Indiferente
I2	Error cuando campo ruta destino vacío	Indiferente	Indiferente
I3	Error cuando campo archivo fuente invalido	Indiferente	Indiferente
I4	Error cuando campo ruta destino invalido	Indiferente	Indiferente
I5	Error cuando archivo fuente no es .uml ni .use	Indiferente	Indiferente
I6	Error cuando archivo USE tiene errores gramaticales	Indiferente	Indiferente
I7	Error cuando archivo XMI tiene errores gramaticales	Indiferente	Indiferente

Tabla 2. Plan de pruebas.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga