FACULTAD DE CIENCIAS

UNIVERSIDAD DE CANTABRIA

Proyecto Fin de Carrera

Evaluación de la viabilidad de un editor visual para el modelo MAST 2 sobre la plataforma Eclipse

(Assessing the feasibility of a visual editor for the MAST 2 model on the Eclipse platform)

Para acceder al título de

INGENIERO EN INFORMÁTICA

Autor: Javier de la Dehesa Cueto-Felgueroso

Julio 2012

FACULTAD DE CIENCIAS

INGENIERÍA EN INFORMÁTICA

CALIFICACIÓN DEL PROYECTO FIN DE CARRERA

Realizado por:	Javier de la Dehesa Cueto-Felgueroso
Director del PFC:	Michael González Harbour
Título:	Evaluación de la viabilidad de un editor visual para el modelo MAST 2 sobre la plataforma Eclipse
Title:	Assessing the feasibility of a visual editor for the MAST 2 model on the Eclipse platform

Presentado a examen el día:

para acceder al Título de

INGENIERO EN INFORMÁTICA

Composición del Tribunal:

Presidente (Apellidos, Nombre):	Michael González Harbour
Secretario (Apellidos, Nombre):	María del Carmen Martínez Fernández
Vocal (Apellidos, Nombre):	Rafael Menéndez de Llano Rozas
Vocal (Apellidos, Nombre):	Marta Elena Zorrilla Pantaleón
Vocal (Apellidos, Nombre):	Roberto Sanz Gil

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: Vocal

Fdo.: Vocal

Fdo.: Vocal

Fdo.: El Director del PFC

Resumen

En este trabajo se evalúa la viabilidad de un nuevo editor visual para MAST 2, la segunda versión del modelo de análisis y simulación de sistemas de tiempo real desarrollado por el grupo CTR de la Universidad de Cantabria.

El objetivo es analizar la posibilidad de reemplazar el antiguo editor independiente, incompatible con la nueva versión, por uno integrado dentro del entorno de desarrollo Eclipse, utilizando para ello el framework GMF de construcción de editores gráficos de modelos.

Para ello, se ha llevado a cabo una implementación parcial del editor, ofreciendo recursos para una potencial ampliación y extrayendo las conclusiones necesarias para conformar una evaluación crítica de la tecnología.

Palabras clave — *mast, eclipse, gmf, emf*

Abstract

This work assesses the feasibility of a visual editor for MAST 2, second version of the real-time systems analysis and simulation model developed by the CTR group at University of Cantabria.

The aim is to analyze the possibility of replacing the former editor independent, incompatible with the new version, by a new one integrated within Eclipse development environment using GMF, a framework to create graphical modeling editors.

For this purpose, a partial implementation of the editor has been carried out, providing resources for potential extensions and drawing the necessary conclusions to form a critical evaluation of the technology.

Keywords — *mast, eclipse, gmf, emf*

Agradecimientos

A Michael, a César y a todo el grupo de CTR por su confianza y su apoyo.

A mi familia, a mi novia y a mis amigos por su paciencia.

Gracias a todos.

Índice general

Índice de figuras

Figura **Página**

Índice de fragmentos de código

Fragmento **Página**

Capítulo 1

Introducción

Entendemos por sistemas de tiempo real [23] aquellos en los que la corrección del cálculo no depende solo de su corrección lógica sino también del tiempo en el que se produce. En [19] se listan algunos casos de uso típicos de sistemas de tiempo real, entre los que se encuentran el control digital, el procesado de señal o las aplicaciones multimedia.

Este tipo de sistemas se modelan en base a un conjunto particular de objetos o clasificadores básicos, como tareas, temporizadores o planificadores. El grupo de Computación y Tiempo Real (CTR) de la Universidad de Cantabria trabaja desde el año 2000 en el proyecto *Modeling and Analysis Suite for real-Time applications* (MAST), un modelo general para sistemas de tiempo real acompañado de un conjunto de herramientas de análisis y simulación. A día de hoy, el grupo trabaja en la segunda versión del modelo, con importantes cambios y mejoras con respecto a la primera [6][14].

El modelo MAST se basa en un formato basado en *eXtensible Markup Language* (XML) de fácil comprensión tanto para máquinas como para personas, y no es difícil describir un modelo de manera puramente textual. Sin embargo, a medida que la complejidad de los modelos crece la interacción directa con el documento XML se vuelve más impracticable. En este punto, el uso de un software de asistencia al modelado resulta imprescindible.

Hasta ahora, MAST ha contado con un editor visual de modelos propio, una aplicación de escritorio desarrollada en Ada [24] y basada en la librería GTK+

[17]. Aun siendo funcional, el editor presenta dos inconvenientes: en primer lugar, está diseñado para la primera versión de MAST, por lo que, en cualquier caso, sería necesario adaptarlo a la nueva versión; en segundo lugar, su concepción se aleja de las tendencias actuales en el ámbito de los entornos de desarrollo, funcionando como un componente aislado en lugar de como una pieza integrada en una plataforma mayor.

En esta coyuntura de cambio de versión y de herramientas, se ha propuesto la evaluación de alternativas a la aplicación existente como software de modelado. En este trabajo se estudia la tecnología *Graphical Modeling Framework* (GMF), dentro de la plataforma Eclipse, como framework de apoyo a dicho editor, proponiendo una implementación parcial del mismo con el objetivo de analizar la viabilidad de un desarrollo completo.

El documento comienza con una vista general del contexto tecnológico que rodea al proyecto: la plataforma Eclipse, el framework GMF y el modelo MAST. A continuación se detalla el trabajo de desarrollo llevado a cabo, esencialmente un editor para una de las facetas del modelo MAST con características añadidas. Para el desarrollador interesado, se incluye una guía de ampliación del editor al resto del modelo MAST, de modo que se integre con los editores ya existentes; por otro lado, se proponen diferentes líneas de trabajo en las que mejorar el editor no desarrolladas en este trabajo. Por último, se exponen las conclusiones, donde se recogen los resultados de la evaluación de la tecnología.

Capítulo 2

Trasfondo tecnológico

2.1. Eclipse

Eclipse es una plataforma de código abierto para la construcción de aplicaciones de escritorio multiplataforma. Comenzó siendo un entorno de desarrollo basado en un producto propietario de IBM [11] para ir evolucionando hacia una plataforma de propósito general, siendo el entorno de desarrollo nada más que un caso particular de aplicación (si bien el más popular).

Eclipse está basado en la arquitectura orientada a plugins Equinox, implementación del estándar *Open Services Gateway initiative* (OSGi) [18], y define el paquete o *bundle* como unidad de software básica. Un paquete es un proyecto empaquetado de Eclipse que incluye un manifiesto en el que se indican las dependencias del mismo y, posiblemente, un plugin. A su vez, un plugin contiene una cantidad arbitraria de extensiones, funcionalidades que se añaden al sistema, y puntos de extensión, que permiten extender la extensibilidad (valga la redundancia) de la plataforma. Cada extensión se define para un punto de extensión, de modo que, por ejemplo, el editor de código Java es una extensión del punto de extensión de editores.

Así, aplicaciones basadas en Eclipse tan diferentes como el entorno de desarrollo para Java, las herramientas de modelado o entorno de desarrollo web (disponibles para descarga en [8]) no son más que diferentes conjuntos de paquetes instalados sobre la misma plataforma común.

2.2. GMF

El proyecto Eclipse GMF [9] es un framework para la construcción de editores visuales sobre la plataforma Eclipse. Está basado en *Eclipse Modeling Framework* (EMF), el framework de modelado de Eclipse, y proporciona herramientas para editar modelos de forma gráfica, a través de diagramas.

Frente a otros frameworks o librerías, GMF ofrece la ventaja de la integración con el entorno de Eclipse, siendo uno más de los servicios ofrecidos para modelos EMF. La figura 2.1 muestra la metodología de trabajo propuesta por GMF [9].

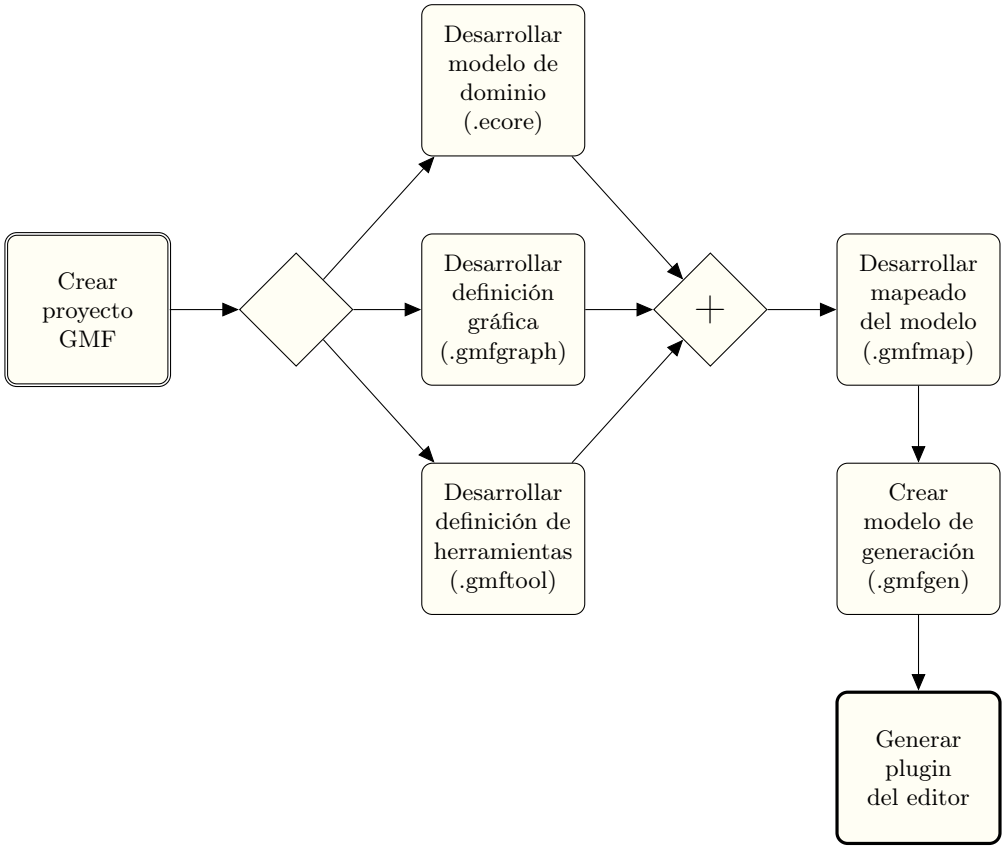

Fig. 2.1: Metodología de trabajo en GMF

El primer paso para crear un editor es abrir un nuevo proyecto de Eclipse. GMF incluye una plantilla de proyecto propia con una cierta estructura y una serie de dependencias configuradas. Sin embargo, cualquier proyecto de Eclipse correctamente configurado es igualmente válido.

Una vez creado el proyecto, el desarrollo comienza con la construcción de una descripción del modelo de dominio del editor. Para ello, EMF provee el metamodelo común Ecore para definir modelos de datos independientes del lenguaje.

En general, los desarrollos basados en EMF utilizan uno o más modelos Ecore mediante las herramientas de edición, instanciación y generación de código provistas por el framework, así como la *Application Programming Interface* (API) de reflexión. El resultado del desarrollo del modelo de dominio es un archivo `.ecore`. Este proceso es independiente de GMF.

La definición gráfica consiste en el diseño de los aspectos visuales del editor, fundamentalmente los nodos, conexiones y etiquetas de las que dispondrá. Es importante señalar que la definición gráfica no está ligada de manera directa al modelo de dominio; es más, ni siquiera es estrictamente necesario conocerlo para desarrollarla (si bien es altamente recomendable). Esto significa que las figuras no tienen por qué corresponderse de manera unívoca con elementos del modelo, sino que pueden ser utilizadas para representar objetos de diferentes clases. La definición gráfica se almacena en un fichero `.gmfgraph`.

La definición de herramientas describe el contenido de la paleta del editor gráfico. Suele constar de uno o más grupos de herramientas con botones para crear cada elemento del modelo presente en el diagrama, aunque puede contener otros tipos de herramientas. La definición de herramientas se almacena en un archivo `.gmftool` y, al igual que la definición gráfica, no depende directamente del modelo de dominio.

Una vez se dispone de un modelo Ecore, una definición gráfica y una definición de herramientas, se construye el mapeado del modelo. Esto no es más que un nexo de unión entre los tres componentes: para cada elemento del modelo que deba estar reflejado en el diagrama se asigna una figura y, en su caso, una herramienta. Define el significado lógico de los nodos, enlaces, etiquetas y botones diseñados previamente, permitiendo además añadir restricciones a las asociaciones. El mapeado se almacena en un archivo `.gmfmap`.

Con la construcción del mapeado del modelo termina el desarrollo necesario para hacer un editor GMF básico. El siguiente paso es obtener el modelo de generación, que se crea automáticamente a partir del mapeado del modelo. El modelo de generación contiene la información necesaria para generar el código del editor GMF, tal como el directorio de generación, el identificador del plugin o los nombres de las clases que se crearán. En el caso general, no es necesario realizar cambios sobre el modelo creado automáticamente. El modelo de generación se almacena en un fichero `.gmfgen`.

A partir del modelo de generación, GMF es capaz de generar un nuevo proyecto con un plugin que contiene el conjunto de extensiones necesario para crear y editar modelos con el editor diseñado. Este plugin puede utilizarse posteriormente de manera local o ser distribuido a través de los mecanismos comunes ofrecidos por Eclipse.

Desde el punto de vista de un desarrollo según el patrón modelo-vista-controlador (MVC) [22], EMF proporciona las herramientas para definir el modelo, mientras GMF facilita la creación de la vista y la parte del controlador relacionada exclusivamente con la interacción entre las anteriores. Queda, por tanto, a responsabilidad del desarrollador la implementación de la lógica de negocio inherente al modelo que no haya quedado cubierta. Aunque GMF no define un mecanismo explícito a este efecto, sí provee cierta flexibilidad en la creación del modelo de generación y la generación automática de código, permitiendo personalizar las plantillas predefinidas usadas internamente en las transformaciones *Model to model* (M2M) y *Model to text* (M2T).

2.3. MAST

MAST [4] es un modelo desarrollado por el grupo de investigación CTR de la Universidad de Cantabria. Permite describir el comportamiento temporal de sistemas de tiempo real para ser analizados mediante técnicas de análisis de planificabilidad. Asimismo, incluye una serie de herramientas de código abierto para la puesta en práctica de dichas técnicas, entre las que se encuentran:

- Análisis de planificabilidad de tiempo de respuesta de peor caso.
- Cálculo de tiempos de bloqueo.
- Análisis de sensibilidad mediante el cálculo de holguras.
- Técnicas de optimización de asignación de prioridades.
- Editor gráfico para la generación de modelos.
- Visor de resultados.

MAST es un trabajo en evolución. Desde la versión inicial se han ido añadiendo nuevas características tanto al modelo como a las herramientas, hasta

llegar a la actual versión estable 1.4.0.1. Sin embargo, el grupo CTR trabaja actualmente en la definición de una segunda versión del modelo más amplia y con importantes diferencias con respecto a la actual. En [6] y [14] se da una visión general del nuevo modelo MAST 2.

Este cambio implica a su vez un rediseño de las herramientas asociadas. En este trabajo, se evalúa la viabilidad de un nuevo editor gráfico sobre una tecnología en particular. Dada la complejidad del modelo MAST (tanto en su versión actual como en la futura), el editor ya existente, incompatible con MAST 2, hace una división del mismo en las siguientes particiones:

- Recursos de procesamiento y planificadores
- Servidores de planificación
- Recursos compartidos
- Operaciones
- Transacciones

Debido a las profundas cambios introducidos en la nueva versión del modelo, MAST 2 no se ajusta bien a esta estructura. No obstante, para diseñar un editor de uso práctico es imprescindible imponer algún tipo de división, ya que el manejo de todos los elementos del modelo (más de cien en la nueva versión) a través de un solo diagrama resultaría imposible.

Dada la naturaleza evaluadora del trabajo, nos centraremos inicialmente en una única partición del modelo representativa; la implementación del editor completo consistiría en una extensión de los mismos conceptos al resto de particiones (ver capítulo 5). Cada partición se correspondería con un editor GMF independiente; por ello, en un punto del desarrollo se añadirá un editor para una nueva partición ficticia, con la intención de estudiar la integración entre los editores de diferentes particiones.

La partición de estudio definida, similar a la de "Recursos de procesamiento y planificadores" del editor existente, abarca los elementos que describen la plataforma sobre la que se ejecuta el sistema de tiempo real modelado. En el siguiente listado se reseñan las clases reflejadas en el diagrama junto a una breve descripción.

`Mast_Model` Es el objeto raíz del modelo, es decir, el "lienzo" del editor. Todos los elementos del diagrama están contenidos directa o indirectamente en una instancia de esta clase. Es la única clase que debe estar representada en cualquier partición posible.

`Regular_Processor` Representa un procesador físico genérico.

`Packet_Based_Network` Red de paquetes con soporte para algún protocolo de tiempo real.

`RTEP_Network` Modela una red que usa el protocolo *Real-Time Ethernet Protocol* (RTEP) [20].
Hereda de `Packet_Based_Network`.

`AFDX_Link` Un enlace entre un procesador y/o un conmutador en una red que usa el protocolo *Avionics Full-DupleX switched ethernet* (AFDX) [16].

`Regular_Switch` Modela un conmutador sencillo.

`AFDX_Switch` Conmutador que funciona de acuerdo a la especificación AFDX.
Hereda de `Regular_Switch`.

`Regular_Router` Enrutador simple con colas de salida *First In, First Out* (FIFO).

`Clock_Synchronization_Object` Mecanismo de sincronización que permite a diferentes recursos de computación compartir un mismo reloj.

`Ticker` Modela un temporizador periódico.

`Alarm_Clock` Temporizador hardware que genera una interrupción en cada sucesión de un evento asociado.

`Primary_Scheduler` Representa el planificador del sistema del recurso de procesamiento asociado.

`Secondary_Scheduler` Planifica la capacidad de cómputo asociada a un recurso planificable a otros recursos.

`Fixed_Priority_Policy` Representa una política de planificación de prioridad fija.

`FP_Packet_Based_Policy` Política de planificación usada por una red de comunicación orientada a paquetes.

`EDF_Policy` Política de planificación *Earliest Deadline First* (EDF), en la que los recursos se planifican en orden de plazo de ejecución más corto.

`AFDX_Policy` Política de planificación usada por redes AFDX.

`Timetable_Driven_Policy` Política de planificación que asigna capacidad de cómputo a los recursos de acuerdo a una tabla de particiones.

`Timetable_Driven_Packet_Based_Policy` Política de planificación basada en tabla de particiones utilizada por una red orientada a paquetes.

`Partition_Window` Cada uno de los elementos de una tabla de particiones.

`Packet_Driver` Modela un controlador de red activado en cada transmisión o recepción de un paquete.

`RTEP_Packet_Driver` Controlador de red que caracteriza el protocolo RTEP. Hereda de `Packet_Driver`.

`Character_Packet_Driver` Especificación de controlador de paquetes con una sobrecarga temporal adicional asociada a la transmisión de cada carácter. Hereda de `Packet_Driver`.

El trabajo llevado a cabo con GMF para implementar este editor permitirá desarrollar una evaluación crítica del framework y de su adecuación al problema.

Capítulo 3

Desarrollo del editor

En los capítulos 1 y 2 hemos dado un escenario de partida para nuestro trabajo. A lo largo de este capítulo describiremos el desarrollo software llevado a cabo para implementar varios de los aspectos más importantes con los que debería contar un editor visual para MAST completo, de cara a una evaluación exhaustiva de la tecnología GMF.

El desarrollo se plantea de manera iterativa a través de una serie de etapas. Para cada etapa se definen unos objetivos y se plantean soluciones. En el capítulo 6 se refieren todos los objetivos que no se han logrado implementar, así como directrices para que el lector interesado pueda ampliar el trabajo.

Los esfuerzos de desarrollo del editor están orientados principalmente a la evaluación de los siguientes aspectos:

- Viabilidad del desarrollo de un editor gráfico para una partición del modelo MAST.
- Viabilidad de integración de varios editores de diferentes particiones.
- Viabilidad de integración del editor o editores con herramientas externas.
- Facilidad de ampliación del trabajo desarrollado.

En el capítulo 5 se detalla la configuración de Eclipse utilizada durante el desarrollo del editor. La compatibilidad con otras configuraciones no está garanti-

zada, si bien las versiones futuras deberían mantener la suficiente compatibilidad hacia atrás.

3.1. Desarrollo de un editor básico para una partición del modelo

3.1.1. Objetivos

Como primera toma de contacto con el framework, implementaremos un editor GMF básico para la partición del modelo MAST descrita en la sección 2.3. El editor debe ser capaz de representar cada elemento de la partición con una figura representativa, esto es, incluyendo los atributos más relevantes del objeto; sin embargo, es necesario poder modificar cualquiera de los atributos a través de algún menú auxiliar. Por otra parte, el diagrama deberá expresar las relaciones de pertenencia y asociación pertinentes.

3.1.2. Desarrollo

Un editor básico GMF puede crearse por completo a partir modelos, es decir, sin necesidad de escribir manualmente ninguna línea de código fuente. Como se explicó en el capítulo 2, un editor básico se construye a partir de un modelo Ecore, una definición gráfica, una definición de herramientas y un mapeado del modelo, dando como resultado un modelo de generación que da lugar al código del editor como tal. En [9] existen varias guías de construcción de editores GMF que cubren la mayoría de los conceptos explicados en esta sección.

Atendiendo a las convenciones de nombres de Eclipse [10], el nombre dado al proyecto GMF creado es `es.unican.mast`. Este proyecto contendrá los modelos manipulados durante el desarrollo, pero no un editor como tal. Estos estarán alojados en diferentes proyectos, por lo general generados automáticamente.

3.1.2.1. Importación del modelo MAST 2

El primer modelo necesario es el modelo Ecore de MAST 2, en el que se describe de manera formal cada uno de los elementos con sus atributos y relaciones. No ha sido necesario crear este modelo, puesto que el grupo CTR ya disponía de él y lo ha ofrecido para este trabajo. Como se ha señalado anteriormente,

MAST 2 es un modelo en construcción; aunque actualmente se trabaja en una versión próxima a la definitiva, periódicamente surgen cambios en el modelo de referencia que pueden no estar reflejados en el modelo Ecore. En general el trabajo desarrollado aquí será válido, pero algunos cambios en el modelo podrían requerir cambios en el editor.

A partir del modelo Ecore importado a la carpeta `model`, `mast.ecore`, se ha creado el modelo `mast.genmodel` con el asistente *EMF Generator Model*. Este nuevo modelo, ligado al anterior, define los detalles necesarios para generar código fuente a partir del modelo Ecore. El modelo generado automáticamente es válido sin modificación alguna, pero es conveniente especificar los nombres de los paquetes de código que se generarán. Para ello, se ha asginado el valor de *Base Package* de `mast.genmodel` a `es.unican`. De este modo, los nombres de los paquetes comienzan siempre por `es.unican.mast`, de manera coherente con el nombre del proyecto.

El siguiente paso es la generación de código a partir del modelo Ecore. Para un editor GMF son necesarios dos componentes: el código del modelo, que construye clases asociadas a cada elemento del modelo con operaciones para acceder a los atributos, y el código de edición, cuyo objetivo es proporcionar facilidades para la manipulación de estas clases y un conjunto de iconos genéricos personalizables. Con `mast.genmodel` abierto en el editor, se ha generado el código con las opciones *Generate Model Code* y *Generate Edit Code* del menú *Generator*, produciendo nuevo código fuente dentro de `es.unican.mast` y un nuevo proyecto `es.unican.mast.edit`.

3.1.2.2. Construcción de la definición gráfica

En este punto, existen dos posibles vías: utilizar los asistentes para la creación de la definición gráfica, la definición de herramientas y el mapeado del modelo o crearlos desde cero. Aunque los asistentes son útiles en una primera toma de contacto con modelos de tamaño medio o reducido, para tamaños mayores resulta más seguro y mantenible controlar manualmente los modelos. Además, los asistentes generan editores formados sólo por figuras básicas sin etiquetas para las propiedades de los elementos del modelo, lo cual no cumple con el objetivo.

Dado que el proyecto está orientado a contener varios editores independientes en el futuro, se ha previsto organizar los modelos bajo un directorio `editor` en

el que se almacenarán carpetas individuales para cada uno. Para esta primera partición de MAST se ha utilizado la carpeta `platform`, dentro de la cual está la definición gráfica `platform.gmfgen`. La definición gráfica consiste en un elemento raíz *Canvas* bajo el cual se agrupan objetos *Figure Gallery*, *Node*, *Connection*, *Compartment* y *Diagram Label*.

`Canvas` representa el "lienzo" sobre el que se dibuja el diagrama. Es la representación del elemento raíz del modelo, en este caso la clase `Mast_Model`. La única propiedad que tiene, *Name*, suele ser un identificador representativo del modelo; sin embargo, como en este caso no abarca todo el modelo, se le ha asignado el valor `Platform`.

Cada elemento *Figure Gallery* puede contener varios *Figure Descriptor*, esto es, definiciones gráficas de figuras. Es necesario como mínimo un *Figure Gallery*, pero puede crearse un número arbitrario de ellos para facilitar la organización visual del modelo. En nuestro caso, se han creado tres: `FirstLevelFigures`, para las figuras que deben mostrarse directamente sobre el lienzo del diagrama, `SecondLevelFigures` para las figuras que deben aparecer contenidas dentro de otras y `Connections` para las conexiones entre objetos.

El diseño de todas las figuras sigue un esquema determinado. Aunque GMF permite reutilizar la misma representación para diferentes clases del modelo ha sido necesario crear manualmente un diseño para cada una, ya que cada figura debe representar los atributos más importantes del objeto que modela. A modo de ejemplo, analizaremos el caso de la clase `Regular_Processor`, mostrado en el diagrama como elemento de primer nivel. La figura 3.1 muestra la estructura del descriptor

Dentro de `FirstLevelFigures` hay un *Figure Descriptor* llamado `Regular-Processor_Desc`, siguiendo una convención general de añadir el sufijo `_Desc` al nombre de la clase para formar el identificación del descriptor. El primer objeto de cada descriptor es un *Rounded Rectangle* de nombre `Container`, que representa un rectángulo de esquinas redondeadas, y, dentro de este, un *Flow Layout* con la propiedad *Vertical* a `true` para forzar una orientación vertical, un *Foreground* para definir el color del borde como `lightGray` y un *Margin Border* con un *Insets* que introduce un margen interno de cinco unidades.

A continuación vienen dos *Label*, `RegularProcessorName` y `Type`, ambos con un elemento *Foreground* para mostrar el texto en color negro y un elemento *Ba-*

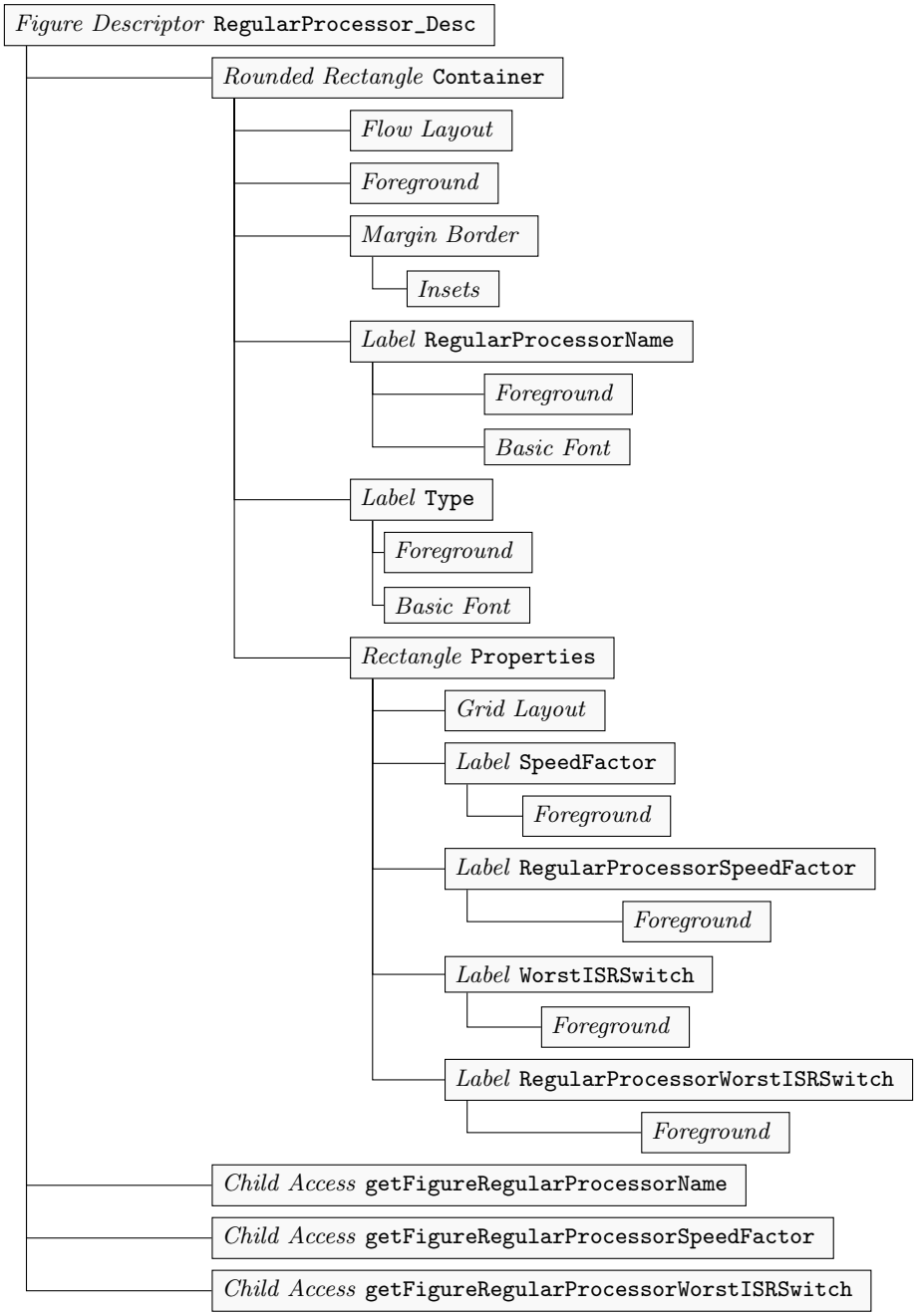

Fig. 3.1: Estructura del descriptor de `Regular_Processor`

sic Font para seleccionar un tamaño de fuente mayor al predefinido (10 para el primero y 9 para el segundo). Representan respectivamente las etiquetas para mostrar el nombre y el tipo de elemento. Las etiquetas con valores variables, es decir, las que muestran atributos propios del objeto (como `RegularProcessor-Name`), deben tener un identificador único; en este caso la convención adquirida ha sido utilizar la concatenación del nombre de la clase y el nombre del atributo en estilo *CamelCase*, es decir, empezando las palabras con mayúscula y

sin separadores. La etiqueta `Type`, en cambio, siempre muestra el mismo valor (en este caso `Regular Procesor`) y su identificador puede repetirse en diferentes descriptores sin conflicto.

El siguiente elemento dentro de `Container` es el *Rectangle* llamado `Properties`, que muestra las propiedades más relevantes del objeto para poder ser editadas directamente sobre la figura. Las propiedades se muestran en forma de parejas nombre-valor, para lo cual se utiliza el *Grid Layout* que permite distribuir los elementos del rectángulo en columnas. De este modo, los cuatro elementos *Label* a continuación describen dos parejas nombre-valor. Las cuatro etiquetas tienen un *Foreground* para seleccionar un color de fuente negro, pero, mientras que `SpeedFactor` y `WorstISRSwitch` tienen los valores literales `Speed factor:` y `Worst ISR switch:`, `RegularProcessorSpeedFactor` y `RegularProcessorWorstISRSwitch` tendrán asignado el valor variable del correspondiente atributo.

Por último, ya fuera de `Properties` y de `Container`, están los cuatro *Child Access*. Estos objetos no tienen una representación gráfica como tal, pero permiten referenciar elementos internos de la figura desde otros puntos del modelo. La asociación de etiquetas a valores de atributos requiere disponer de un *Child Access* a la etiqueta, por lo que es necesario uno por cada etiqueta que vaya a mostrar información variable. Aunque los identificadores de los *Child Access* se pueden personalizar, se ha mantenido el valor por defecto propuesto por GMF, el nombre de la figura asociada precedido por `getFigure`.

Todas las figuras de primer nivel en el diagrama siguen este mismo patrón: un rectángulo de esquinas redondeadas encabezado por el nombre y el tipo de objeto, seguidos por las propiedades más relevantes descritas en forma de parejas nombre-valor. En el caso de las figuras de segundo nivel, agrupadas bajo el *Figure Gallery* `SecondLevelFigures`, el patrón es el mismo salvo por el nombre, que no existe como atributo en ninguno de los elementos y por lo tanto no aparece en la figura.

Por último, se han diseñado cuatro conectores bajo la galería `Connections`, una para cada una de las relaciones posibles entre objetos de la partición. Las figuras de conectores son mucho más sencillas que las de nodos. A modo de ejemplo, la figura 3.2 muestra la estructura del conector que debe unir un `Regular_Processor` a un `Ticker` o un `Alarm_Clock` a través del atributo `System_Timer`.

En el caso de los conectores se han identificado con el nombre de la clase y el

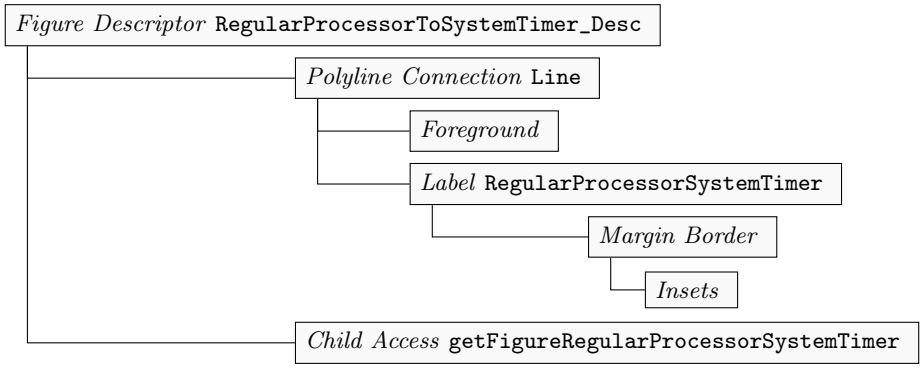

Fig. 3.2: Estructura del descriptor de la referencia `System_Timer`

nombre del atributo unidos por la partícula `To`, añadiendo, al igual que antes, el sufijo `_Desc`. El primer elemento del descriptor es el *Polyline Connection* `Line`, que contiene a su vez un *Foreground* para configurar el color del conector y un *Label* para mostrar sobre la línea texto que discrimine la relación entre los elementos (por ejemplo, para diferenciar si un `Ticker` conectado a un `Regular_Processor` es el temporizador del sistema o un temporizador adicional). El *Margin Border* y el *Insets* añaden un margen inferior de quince unidades para forzar al texto a estar por encima de la conexión, en caso de ser horizontal.

Aunque los *Label* de todos los conectores muestran un valor constante (`System timer` en este caso) es necesario crearlos con un identificador único ya que, como veremos más adelante, deberán referirse desde otros puntos del modelo para ser mostrados correctamente. Al igual que antes, se ha utilizado la unión del nombre de la clase y el atributo que modela la conexión. Por último, el *Child Access* del final está asociado a la etiqueta `RegularProcessorSystemTimer` para poder acceder posteriormente. El resto de las conexiones siguen la misma estructura, cambiando únicamente el valor de la etiqueta y de los identificadores.

El resto de elementos de la definición gráfica no describen objetos gráficos de por sí, sino que los encapsulan para poder ser posteriormente asociados con entidades lógicas en el mapeado del modelo. Un *Node* representa un gráfico que puede posicionarse libremente en el diagrama y, en principio, sin dependencias con otros. Por ejemplo, el nodo `RegularProcessor` tiene asociado el descriptor `RegularProcessor_Desc`, de modo que la clase asociada a dicho nodo estará representada por esta figura.

Del mismo modo, cada *Connection* es un objeto que permite asociar una figura a una relación entre clases. Aunque no es obligatorio, las conexiones

suelen estar referidas a figuras lineales. Continuando con el ejemplo, la conexión `RegularProcessorToSystemTimer` será la que permita mapear la figura `RegularProcessorToSystemTimer_Desc`.

Los *Compartment* definen compartimentos que permiten alojar nodos dentro de otros nodos cuando exista una relación de pertenencia. Por ejemplo, cada `Primary_Scheduler`, contiene una política de planificación, por lo que se ha creado el compartimento `PrimarySchedulerPolicy` asociado al descriptor correspondiente.

Finalmente, hay un *Diagram Label* por cada etiqueta que deba mostrar el valor de un atributo de un objeto, así como por cada etiqueta mostrada sobre un conector. Cada *Diagram Label* está asociado tanto al descriptor que contiene la etiqueta como al *Child Access* correspondiente. Así, `RegularProcessorName` se refiere a `RegularProcessor_Desc` y a `getFigureRegularProcessorName`. Además, todas las etiquetas que muestran el nombre de un objeto tienen la propiedad *Element Icon* con valor `true`, por lo que en el diagrama aparecen junto al icono del objeto.

3.1.2.3. Construcción de la definición de herramienttas

El segundo modelo creado es la definición de herramientas. En este se describen las herramientas de las que dispondrá el editor, no sólo los botones de la paleta sino también barras de herramientas, menús principales o menús contextuales. Para nuestro editor sólo se ha construido una paleta con herramientas de creación de objetos, organizados en seis *Tool Group*:

Processing resources Recursos de procesamiento como procesadores, redes y elementos de red.

Timing objects Objetos de sincronización y temporización.

Schedulers Planificadores primarios y secundarios.

Scheduling policies Políticas de planificación.

Drivers Controladores de red.

Connections Asociaciones entre objetos del modelo.

Cada herramienta de la paleta es un *Creation Tool* con el texto que debe mostrar y que a su vez contiene dos *Default Image*, un objeto sin propiedades que indica que la herramienta utilizará las imagenes (grande y pequeña) creadas automáticamente por EMF para la clase asociada.

3.1.2.4. Construcción del mapeado del modelo

El último modelo a crear en esta etapa es el mapeado del modelo, en el que se expresa la relación entre el modelo Ecore, la definición gráfica y la definición de herramientas.

Para nuestro editor, el mapeado se expresa con tres tipos de objetos. En primer lugar un *Canvas Mapping*, único elemento obligatorio de este modelo, en el que se indican los otros tres modelos que intervienen. En este caso, el Ecore sería común para todas las particiones, mientras que la definición gráfica y la definición de herramientas serían particulares de cada una. Además, especifica la clase raíz del modelo, que en este caso es `Mast_Model`.

El segundo tipo de objeto son los *Top Node Reference*, que representan un mapeado de una clase a un objeto de primer nivel en el diagrama. Dentro de cada uno de estos hay un *Node Mapping*, con la información propia del mapeado. La figura 3.3 muestra la estructura de la información de mapeado para la clase `Packet_Based_Network`.

La raíz *Node Mapping* contiene la referencia a la clase mapeada, al nodo correspondiente definido en la definición gráfica y al botón de la definición de herramientas. El primer elemento es un *Constraint* con la restricción *Object Constraint Language* (OCL) `self.oclIsTypeOf(Packet_Based_Network)`. Este tipo de restricción solo es necesaria para clases del diagrama con subclases que también aparecen en el diagrama, en cuyo caso es necesario especificar que el mapeado solo se aplica a objetos de la superclase; de no señalarlo, un objeto de una subclase tendría dos mapeados asignados, causando un mal funcionamiento en el editor.

Para cada propiedad del objeto mostrado en el nodo del diagrama existe un *Feature Label Mapping* en el que se indica la propiedad y la etiqueta que la representa en el diagrama. Todas las figuras de primer nivel tienen al menos una etiqueta para mostrar la propiedad `Name`.

El resto de elementos definen los subnodos que admite la clase. Por cada

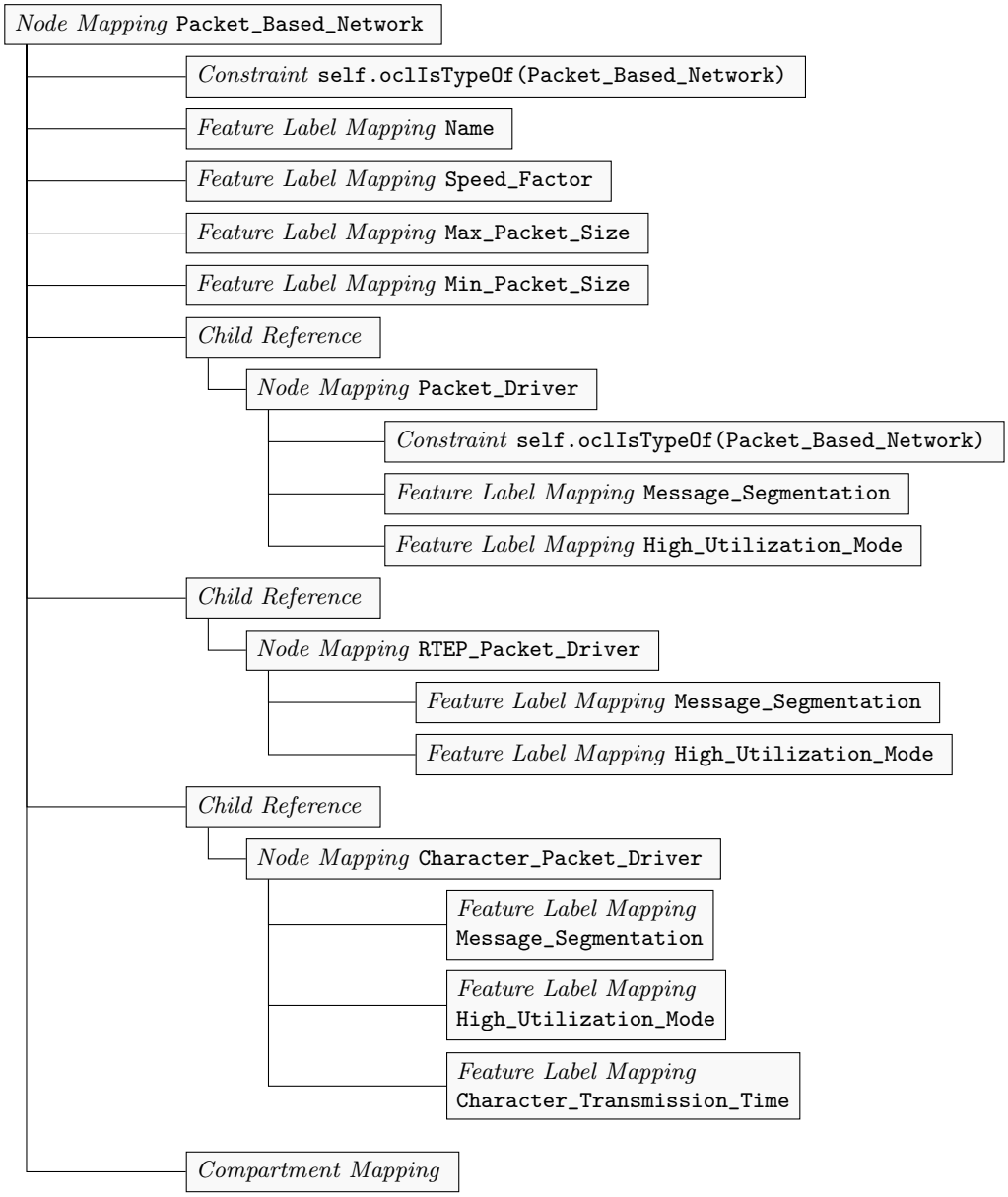

Fig. 3.3: Estructura del mapeado para la clase `Packet_Based_Network`

tipo de subnodo existe un *Child Reference*, todos ellos asociados al *Compartment Mapping* que los contiene. Cada uno contiene un nuevo *Node Mapping* que describe el mapeado de cada subnodo. De nuevo, para `Packet_Driver` se define un *Constraint* para evitar aplicar el mapeado a las subclases `RTEP_Packet_Driver` y `Character_Packet_Driver`, descritas a continuación. Asimismo, a cada propiedad mostrada en el subnodo le corresponde un `Feature Label Mapping`.

Por último, el mapeado del modelo tiene también una serie de *Link Mapping*, cada uno de los cuales define una relación entre objetos del diagrama. De manera similar a los nodos, asocian una relación del modelo Ecore con una conexión de

la definición gráfica y un botón de la definición de herramientas. Los enlaces se definen siempre globalmente, tanto para nodos de primer nivel como para subnodos, y el único elemento que contienen es un *Feature Label Mapping* para la etiqueta del enlace.

3.1.2.5. Generación del editor

Una vez desarrollados todos los modelos, la generación del editor es automática. En primer lugar es necesario crear el modelo de generación a través de la opción correspondiente del menú contextual del mapeado del modelo, con la configuración predeterminada. El modelo resultante contiene toda la información necesaria para generar un editor GMF válido; sin embargo, de cara a una mejor organización, se han modificado algunas de las propiedades del elemento *Gen Editor Generator*. Dado que el editor completo incorporaría un editor por cada partición del modelo, es necesario definir una identificación diferente para los diagramas generados por cada uno. A tal efecto, *Diagram File Extension* se ha configurado a `mast_platform` y *Model ID* a `Mast Platform`. Asimismo, el código generado debe alojarse en un proyecto propio, por lo que a *Editor Plug-in Directory* se le ha dado el valor `/es.unican.mast.editor.platform/src` y a *Package Name Prefix* `es.unican.mast.editor.platform`.

Hecho esto, el código del editor se crea con el comando *Generate diagram code* del menú contextual del modelo de generación. El proyecto `es.unican.mast.editor.platform` producido puede probarse ejecutando una instancia de prueba de Eclipse desde el propio entorno de desarrollo o ser exportado como plugin para su distribución.

3.2. Ampliación de las funcionalidades del editor

3.2.1. Objetivos

El editor desarrollado en el apartado 3.1 cubre los aspectos básicos de GMF y permite describir la partición de la plataforma del modelo correctamente mediante nodos, subnodos y conexiones. Sin embargo, no se han tenido en cuenta elementos de mayor profundidad en el modelo; en particular, las clases `Timetable_Driven_Policy` y `Timetable_Driven_Packet_Based_Policy` tienen un atributo

`Partition_Table` que contiene una lista de objetos `Partition_Window`, y que no puede modificarse directamente desde la interfaz del editor.

Aunque es posible anidar nodos a profundidad arbitraria, el uso de más de dos niveles impacta negativamente sobra la usabilidad del editor. Además, en el caso particular de la propiedad `Partition_Table` se trata de una colección que puede llegar a tener una gran cantidad de elementos, cada uno de los cuales contiene información sencilla. En esta segunda etapa, por tanto, se soluciona este problema de un modo general, simplificando la resolución de posibles casos similares en editores de diferentes particiones del modelo.

3.2.2. Desarrollo

Para soportar la edición del campo `Partition_Table` se ha optado por desarrollar una sección adicional dentro de la vista de propiedades para las clases `Timetable_Driven_Policy` y `Timetable_Driven_Packet_Based_Policy`; de hecho, es suficiente con definirla sólo para la primera, ya que la segunda es una subclase y puede heredar la compatibilidad.

La nueva sección muestra una tabla editable con la lista de elementos contenidos en la propiedad modelada y con una columna por cada propiedad de los objetos de la lista. Además, cuenta con un botón para añadir nuevos elementos y otro para eliminarlos, y se ordena automáticamente según el valor de la primera columna.

La mayor parte de la funcionalidad está implementada en la clase Java abstracta `AbstractEditableCollectionProperty`, planteada para ser heredada cada vez que sea necesaria una extensión de este tipo — i. e. una subclase por cada propiedad representada de este modo. Las responsabilidades de esta clase son:

- Construir la interfaz de la extensión a partir de componentes de la librería gráfica de Eclipse *Standard Widget Toolkit* (SWT).
- Extraer la información del modelo a partir del objeto del diagrama y mostrar la información de colección modelada en la interfaz.
- Recoger los eventos producidos por el usuario y actuar en consecuencia, añadiendo, modificando y eliminando objetos de la colección.

- Llevar a cabo las operaciones sobre el modelo a través de la pila de comandos del dominio de edición, permitiendo deshacer y rehacer la acción de manera natural.

- Mantener una sincronización adecuada entre la interfaz y el modelo.

Por su parte, cada subclase debe implementar los siguientes métodos:

`getCollectionFeature()` Devuelve `EStructuralFeature` correspondiente a la colección modelada, obtenida de manera trivial a través de las clases generadas automáticamente por EMF a partir del modelo Ecore.

`addBindings()` Añade los enlaces entre el modelo y la vista llamando al método `addBinding(String,EAttribute)`, en el que se indica la cabecera de la columna y el atributo que mostrará.

La clase `PartitionTablePropertySection` implementa estos métodos para el caso de la propiedad que nos ocupa.

Para que la extensión sea aplicable sólo a ciertos tipos de objetos del diagrama es necesaria una clase de filtrado. La clase `ModelClassFilter` ha sido desarrollada a tal efecto, de manera que cada extensión particular solo necesitará implementar el método `filteredClass()` para devolver la clase compatible con ella. En el caso de `Partition_Table`, el filtro está implementado en la clase `TimetableDrivenFilter`.

Por último, es necesario añadir la extensión correspondiente al editor. Para ello, se ha añadido una *Custom Property Tab* al *Property Sheet* del modelo de generación con el nombre `Partition table`. En esta se especifica la clase `PartitionTablePropertySection` como clase de implementación de la pestaña de propiedades y la clase `TimetableDrivenFilter` como filtro, a través de un *Custom filter*. Al regenerar el proyecto, el nuevo editor dispone de la nueva sección de propiedades.

De este modo se resuelve el modelado de la propiedad `Partition_Table` con la posibilidad de reutilizar la implementación en casos similares. En el capítulo 5 se muestra el uso de estas características desde el punto de vista del desarrollador de un nuevo editor con ejemplos de código.

3.3. Integración y extensibilidad para nuevos editores de diferentes particiones del modelo

3.3.1. Objetivos

El siguiente paso en el desarrollo es facilitar la integración de editores de diferentes particiones del modelo, de modo que el futuro desarrollador se limite en lo posible a la definición de los aspectos propios de la partición en particular (tal y como se describe en el capítulo 5).

Para ello es necesario introducir algunas modificaciones en las plantillas de generación de código, así como desarrollar nuevos asistentes integrados.

3.3.2. Desarrollo

Para este tramo del desarrollo se ha creado otro editor a efectos de prueba. Se ha planteado una posible partición del modelo para la descripción de las operaciones del sistema de tiempo real, pero, por simplicidad, solo se ha incluido la clase `Message`, una de las más sencillas de entre las que debieran estar contenidas en la partición. La construcción de la definición gráfica, la definición de herramientas y el mapeado del modelo se ha llevado a cabo de manera similar a como se explicó en la sección 3.1.

3.3.2.1. Modificación de las transformaciones de creación del modelo de generación

El modelo de generación creado por defecto es correcto para un editor GMF aislado, pero no para nuestros objetivos. Para modificar este comportamiento se han adaptado las transformaciones en lenguaje QVTO [13] responsables del proceso, partiendo de las originales ubicadas en la carpeta `transforms` del plugin `org.eclipse.gmf.bridge`. La versión modificada, en el directorio `transforms/-gmfgen` del plugin `es.unican.mast`, añade a la transformación `Map2Gen` (fichero `Map2Gen.qvto`) el fragmento 3.1.

Este fragmento utiliza el nombre del *Canvas* en la definición gráfica para identificar de manera unívoca la partición del modelo y su editor. El primer grupo de instrucciones asigna valores adecuados para las propiedades del modelo de generación (`genEditor`) que deben ser únicas para cada editor: la extensión de

```
[...]
-- Set partition custom properties
var partitionName  : EString :=
    mapRoot.diagram.diagramCanvas.name;
var partitionTrim  : EString :=
    partitionName.toLower().replace(" ", "");
genEditor.diagramFileExtension :=
    ("mast_" + partitionTrim).toLower();
genEditor.packageNamePrefix :=
    ("es.unican.mast.editor." + partitionTrim).toLower();
genEditor.modelID := "Mast " + partitionName;
genEditor.pluginDirectory := ("/es.unican.mast.editor." +
    partitionTrim + "/src").toLower();
genEditor.plugin.iD := ("es.unican.mast.editor."
    + partitionName).toLower();
genEditor.plugin.activatorClassName := "Mast" +
   partitionTrim
    + "DiagramEditorPlugin";
genEditor.plugin.name := "Mast " + partitionName + "
   editor";

-- Set dynamic templates
genEditor.dynamicTemplates := true;
genEditor.templateDirectory :=
    "/es.unican.mast/templates/gmf";

-- shared editing domain
genEditor.diagram.editingDomainID :=
    "es.unican.mast.editor.EditingDomain";

-- plugin properties
genEditor.plugin.provider := "Universidad de Cantabria";
[...]
```

Fragmento 3.1: Cambios añadidos a la transformación QVTO del modelo de generación

los archivos de diagramas (`diagramFileExtension`), el prefijo para los nombres de los paquetes (`packageNamePrefix`), el identificador del modelo (`modelID`), el directorio donde se genera el código (`pluginDirectory`), el identificador del plugin (`plugin.iD`), el nombre de la clase de activación del plugin (`plugin.activatorClassName`) y el nombre del plugin (`plugin.name`). Las siguientes dos propiedades, `dynamicTemplates` y `templateDirectory`, configuran el uso de plantillas dinámicas de generación de código, referidas más adelante. `diagram.editingDomainID` está relacionado con la integración del dominio de edición de los editores (ver capítulo 6), y `plugin.provider` simplemente señala la organización responsable del plugin.

Además, en la misma transformación, se ha comentado la instrucción in-

dicada en el fragmento 3.2 para evitar la creación del elemento *Navigator*, que construye un navegador del modelo en forma de árbol sobre la vista de archivo. Aunque puede ser útil, al crear varios editores se generan múltiples navegadores, lo cual resulta confuso.

```
[...]
--if not rcp then new Navigator(mapModel, gmfgenModel)
--  .transform() endif;
[...]
```

Fragmento 3.2: Eliminación del navegador del modelo de generación

Por último, se han practicado dos pequeñas modificaciones sobre la transformación `PropertySheet` en el fichero `PropertySheet.qvto`, tal y como se indica en el fragmento 3.3.

```
[...]
mapping GMFMAP::CanvasMapping::propertySheet
    (editorGen : GMFGEN::GenEditorGenerator)
    : GMFGEN::GenPropertySheet {
    --result.tabs += object GMFGEN::GenStandardPropertyTab
    --  { iD := "appearance" };
    --result.tabs += object GMFGEN::GenStandardPropertyTab
    --  { iD := "diagram" };
    [...]
}
[...]
```

Fragmento 3.3: Personalización de las pestañas de propiedades

Las líneas comentadas evitan la creación de pestañas adicionales de personalización de estilos en la vista de propiedades, favoreciendo la usabilidad en los casos en que se utilizan pestañas adicionales creadas mediante el método implementado en la sección 3.2.

Para crear el modelo de generación de acuerdo a la nueva transformación sólo es necesario señalar la opción *Use QVTO transformation* en la última página del asistente, indicando la ruta del fichero con la transformación principal, i. e. `platform:/resource/es.unican.mast/transforms/gmfgen/Map2Gen.qvto`.

3.3.2.2. Integración de asistentes de creación de diagramas

Como parte de la integración de los editores, se han creado asistentes comunes para la creación de diagramas. Sin embargo, de manera predeterminada cada

nuevo editor GMF crea, junto con las extensiones del editor propiamente dicho, extensiones que añaden un asistente para la creación de diagramas y una opción en el menú contextual del modelo para ejecutarlo. Para evitar la generación de estas extensiones se han utilizado plantillas dinámicas de GMF.

Las plantillas dinámicas permiten personalizar el código producido por el framework de manera sencilla, tal como se explica en [21]. Las plantillas predeterminadas, escritas mayoritariamente en lenguaje Xpand [**Xpand07**], están en el directorio `templates` del plugin `org.eclipse.gmf.codegen`. Gracias a las características de orientación a aspectos de Xpand, es posible modificar únicamente las plantillas necesarias en lugar de todas.

En el directorio `templates/gmf/aspects/xpt/editor` del plugin `es.unican.mast` se ha creado la plantilla `extensions.xpt`, copia de la original, con los cambios expuesto en el fragmento 3.4.

```
1  «AROUND extensions FOR gmfgen::GenEditorGenerator»
2      [...]
3      <!--
4      <extension point="org.eclipse.ui.newWizards"
5          id="creation-wizard">
6          [...]
7      </extension>
8      -->
9
10 «IF diagram.generateInitDiagramAction()-»
11     <!--
12     «IF null = application»
13     <extension point="org.eclipse.ui.popupMenus"
14         id="init-diagram-action">
15         [...]
16     </extension>
17     «ELSE-»
18     <extension point="org.eclipse.ui.actionSets"
19         id="init-diagram-action">
20         [...]
21     </extension>
22     «ENDIF-»
23     -->
24 «ENDIF-»
25
26 «ENDAROUND»
```

Fragmento 3.4: Modificación de la plantilla de generación de extensiones

La plantilla `extensions.xpt` describe parte de las extensiones añadidas al plugin creado por el modelo de generación. El punto clave de este fragmento son las marcas de comentario XML en las líneas 3, 8, 11 y 23, que deshabilitan

las extensiones que rodean: la primera, identificada con `creation-wizard`, el asistente de creación de diagramas; las dos siguientes, ambas identificadas con `init-diagram-action`, la opción del menú contextual y la acción asociada para iniciar dicho asistente desde una instancia del modelo.

Para los nuevos asistentes comunes se ha creado el plugin `es.unican.mast.editor`. En su clase de activación, `MastEditorPlugin`, se ha creado el método `addEditors()` en el que se especifican los editores existentes a través del método `MastEditorUtil.addEditor(String)` y el nombre de la partición del modelo. Para los dos editores creados, el método queda como muestra el fragmento 3.5.

```
1  /**
2   * Add the editors to be used.
3   */
4  private void addEditors() {
5      // XXX Add here any new editor.
6      MastEditorUtil.addEditor("Platform");
7      MastEditorUtil.addEditor("Operations");
8  }
```

Fragmento 3.5: Método `addEditors` de `MastEditorPlugin`

Utilizando esta información, las clases `MastCreationWizard` y `MastNewDiagramFilesWizard` implementan respectivamente un asistente de creación de un modelo MAST junto con sus diagramas y un asistente de creación de diagramas para un modelo ya existente. Además, la clase `MastInitDiagramFilesAction` implementa una acción de menú para abrir el segundo asistente desde el menú contextual de un modelo. El fragmento 3.6 muestra las extensiones correspondientes añadidas al archivo `plugin.xml`.

La primera extensión registra el asistente de creación de modelos MAST para ser incluido en la lista de asistentes de Eclipse, bajo la categoría *Mast*. La segunda registra la acción de inicialización de diagramas para añadirla al menú contextual de los archivos con extensión `.mast`, que son las instancias del modelo.

3.4. Interoperabilidad con software existente

3.4.1. Objetivos

Como se explicó en el capítulo 2, MAST no es sólo un modelo para definir sistemas de tiempo real, sino también un conjunto de herramientas software de

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
    <!-- New wizard -->
    <extension
        point="org.eclipse.ui.newWizards">
        <category
            name="Mast"
            id="es.unican.mast">
        </category>
        <wizard
            name="%new.name"
            icon="icons/obj16/mast_logo_trans.png"
            category="es.unican.mast"
            class="es.unican.mast.editor.wizard.create
                .MastCreationWizard"
            id="es.unican.mast.editor.wizards
                .MastCreationWizard">
            <description>
                %new.description
            </description>
        </wizard>
    </extension>
    <!-- Initialize diagrams -->
    <extension point="org.eclipse.ui.popupMenus">
        <objectContribution
            id="es.unican.mast.editor.InitDiagram"
            nameFilter="*.mast"
            objectClass="org.eclipse.core.resources.IFile">
            <action
                label="%initDiagramsActionLabel"
                class="es.unican.mast.editor.ui.action
                    .MastInitDiagramFilesAction"
                menubarPath="additions"
                enablesFor="1"
                id="es.unican.mast.editor.ui.action
                    .InitDiagramFilesAction">
            </action>
        </objectContribution>
    </extension>
</plugin>
```

Fragmento 3.6: Extensiones de asistentes integrados

análisis y simulación de los mismos. Estas herramientas operan con modelos MAST descritos en un formato XML determinado, no directamente compatible con el formato de los modelos EMF.

Por ello, es necesario implementar asistentes tanto para transformación de modelos EMF a modelos MAST XML como para la transformación inversa.

3.4.2. Desarrollo

Esta etapa se enmarca en el área de las transformaciones M2M. Las instancias de modelos Ecore de EMF están descritas internamente en *XML Metadada Interchange* (XMI), mientras que MAST utiliza una variante de XML propia. El problema se abordó inicialmente empleando la tecnología *eXtensible Stylesheet Language Transformations* (XSTL), al existir un mecanismo sencillo para transformar el XMI en XML. Sin embargo, la complejidad del procedimiento inverso era elevada.

Acudiendo al trabajo realizado por el grupo de CTR, en [5] se se desarrolla una transformación del modelo MAST descrito en XMI a XML descrito en XMI (modelo EMF de un documento XML). A partir de este se puede realizar la extracción a formato XML de manera automática. De manera análoga, se desarrolló la transformación inversa a partir del XML inyectado a XMI. Se han utilizado estas transformaciones para implementar la funcionalidad de importación y exportación de modelos.

Las transformaciones desarrolladas por el grupo de CTR están escritas en ATL [7], un lenguaje de transformación de modelos con amplio soporte en Eclipse. Sin embargo, ATL está orientado a ser utilizado de manera directa desde el propio entorno de Eclipse, no desde el código de un programa o un plugin. Para llevar a cabo las transformación de manera transparente para el usuario del plugin, se ha desarrollado la clase `MastModelTransformer`. Esta clase utiliza las transformaciones `mast2xml.atl` y `xml2mast.atl` en la carpeta `atl` del proyecto `es.unican.mast` (más concretamente sus versiones compiladas, con extensión `asm`) junto con la clase `AtlLauncher`, disponible entre las herramientas de ATL para Eclipse.

Por otro lado, se han desarrollado dos asistentes, para la exportación y la importación de modelos, en las clases `MastExportWizard` y `MastImportWizard` respectivamente. En el fragmento 3.7 se muestran las correspondientes extensiones añadidas al plugin `es.unican.mast.editor`

Con esto se completa la última fase del trabajo de desarrollo del editor GMF para MAST.

```
[...]

<!-- Import wizard -->
<extension
    point="org.eclipse.ui.importWizards">
    <category
        id="es.unican.mast"
        name="Mast">
    </category>
    <wizard
        category="es.unican.mast"
        class="es.unican.mast.editor.wizard.importer
            .MastImportWizard"
        icon="icons/obj16/mast_logo_trans.png"
        id="es.unican.mast.editor.ui.importer
            .MastImportWizard"
        name="%import.name">
        <description>%import.description</description>
    </wizard>
</extension>

<!-- Export wizard -->
<extension
    point="org.eclipse.ui.exportWizards">
    <category
        id="es.unican.mast"
        name="Mast">
    </category>
    <wizard
        category="es.unican.mast"
        class="es.unican.mast.editor.wizard.exporter
            .MastExportWizard"
        icon="icons/obj16/mast_logo_trans.png"
        id="es.unican.mast.editor.ui.exporter
            .MastExportWizard"
        name="%export.name">
        <description>%export.description</description>
    </wizard>
</extension>

[...]
```

Fragmento 3.7: Extensiones de asistentes de exportación e importación

Capítulo 4

Instalación y uso del editor

En este capítulo se describe como utilizar los plugins desarrollados en el capítulo 3 para modelar los aspectos de MAST abarcados por el editor.

4.1. Instalación

4.1.1. Requisitos

Los plugins se han desarrollado bajo y para la plataforma *Eclipse Modeling Tools Indigo SR2*. Además, requiere los siguientes paquetes:

- `org.antlr.runtime`, versión mayor o igual que `3.0.0` y menor que `3.1.0`.
- `org.eclipse.m2m.atl.common`
- `org.eclipse.m2m.atl.core`
- `org.eclipse.m2m.atl.core.emf`
- `org.eclipse.m2m.atl.drivers.emf4atl`
- `org.eclipse.m2m.atl.dsls`
- `org.eclipse.m2m.atl.engine.vm`

Para una instalación nueva de Eclipse, lo más sencillo es instalar las herramientas de ATL desde el menú de instalación de componentes de modelado (*Help → Install Modeling Components*).

4.1.2. Instalación de los plugins

El editor se compone de cuatro plugins distribuidos en formato JAR:

- `es.unican.mast`
- `es.unican.mast.editor`
- `es.unican.mast.editor.platform`
- `es.unican.mast.editor.operations`

Para instalarlos sólo hay que copiar los archivos JAR a la carpeta `plugins` del directorio de instalación de Eclipse.

4.2. Manual de uso

4.2.1. Creación de un nuevo modelo MAST

La forma habitual de crear un modelo MAST desde cero es utilizando el asistente *Mast model* (figura 4.1), ubicado bajo la categoría *Mast* de la ventana de asistentes de creación (*File* → *New* → *Other...*).

La primera página del asistente permite seleccionar la ubicación y el nombre del fichero que almacenará el modelo MAST. El resto de páginas, opcionales, seleccionan la ubicación de los ficheros de diagramas (uno por cada uno de los editores). Una vez finalizado el asistente se creará el fichero con un modelo MAST vacío junto con los ficheros de diagramas.

4.2.2. Importación de un modelo en XML

Para trabajar con modelos MAST ya existentes descritos en XML es necesario importarlos previamente. Para ello, desde la ventana de asistentes de importación (*File* → *Import...*), bajo la categoría MAST, se encuentra el asistente *Mast XML file* (figura 4.2).

El asistente de importación crea únicamente el fichero del modelo. Para generar los diagramas es necesario seleccionar la opción *Initialize Mast diagrams...* del menú contextual del modelo, como muestra la figura 4.3. El asistente, similar al de creación de modelos, crea los ficheros de diagramas para cada editor.

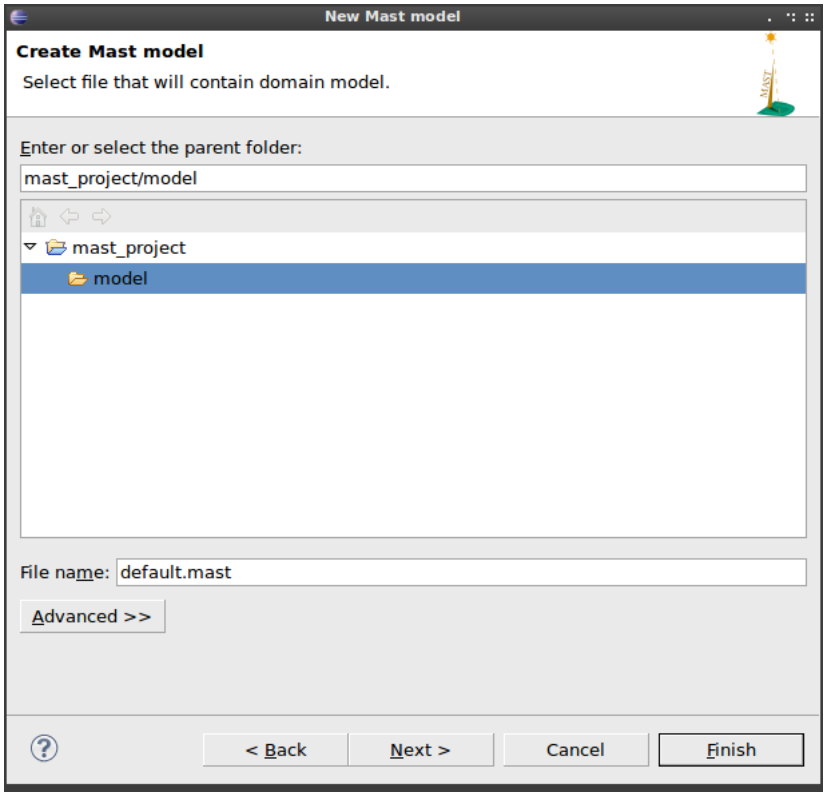

Fig. 4.1: Asistente de creación de modelos MAST

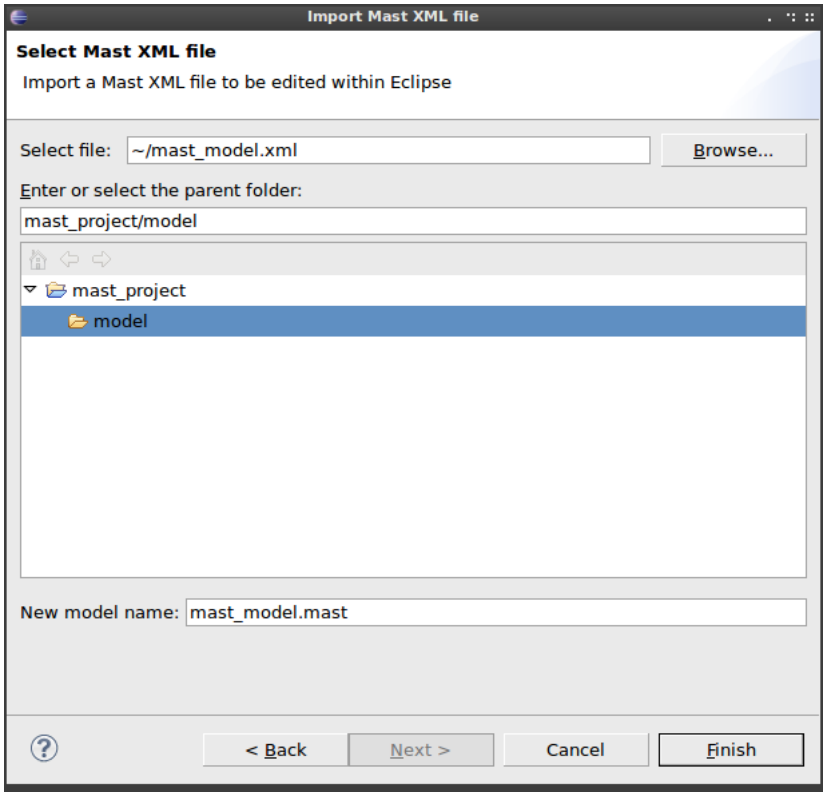

Fig. 4.2: Asistente de importación de modelos MAST en XML

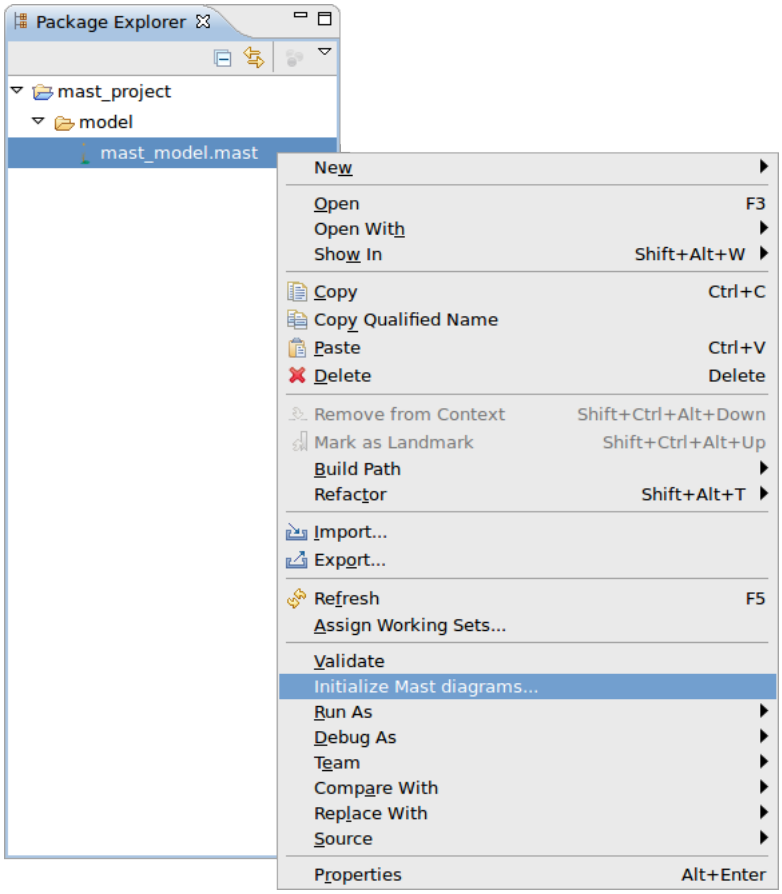

Fig. 4.3: Menú de generación de diagramas

4.2.3. Edición de modelos MAST

Cada modelo MAST en Eclipse se compone de un archivo de modelo y posiblemente de archivos de diagramas. El archivo de modelo se puede editar directamente a través del editor reflexivo de EMF, que ofrece una vista jerárquica del modelo y permite configurar las propiedades y relaciones de cada elemento (figura 4.4). Este editor es compatible con cualquier modelo de EMF.

Para la edición visual del modelo son necesarios los archivos de diagramas. Al abrir el archivo se muestra un editor con todos los nodos incluidos en la partición del modelo correspondiente. En la figura 4.5 se muestra el editor de la plataforma del modelo MAST.

El área principal del editor es el lienzo, sobre el cual se sitúan los nodos que representan los objetos del modelo. A la derecha está la paleta, divida en varios grupos, con los elementos que se pueden poner en el diagrama. El editor sólo permite añadir nodos en los contextos adecuados: por ejemplo, los objetos de primer nivel, como los procesadores, las redes o los planificadores se pueden

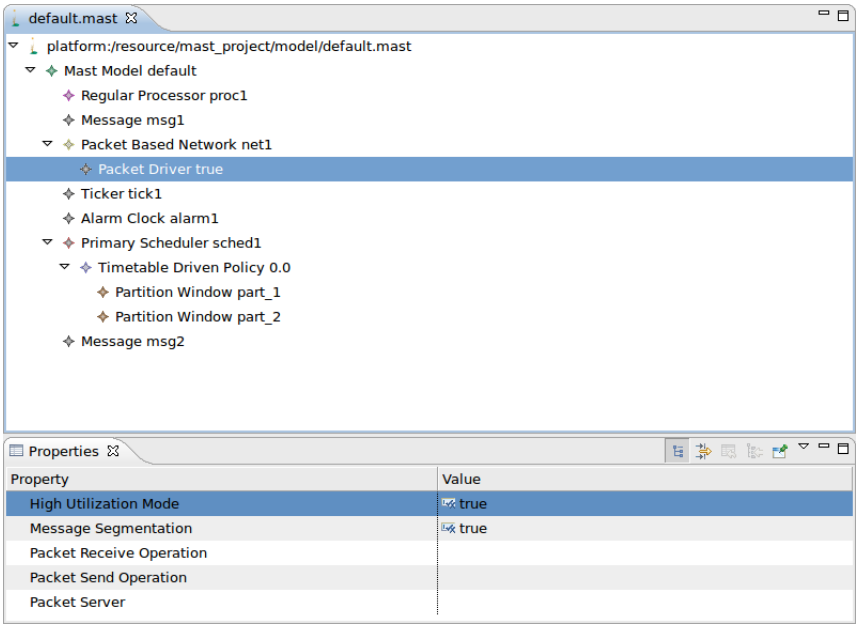

Fig. 4.4: Editor reflexivo

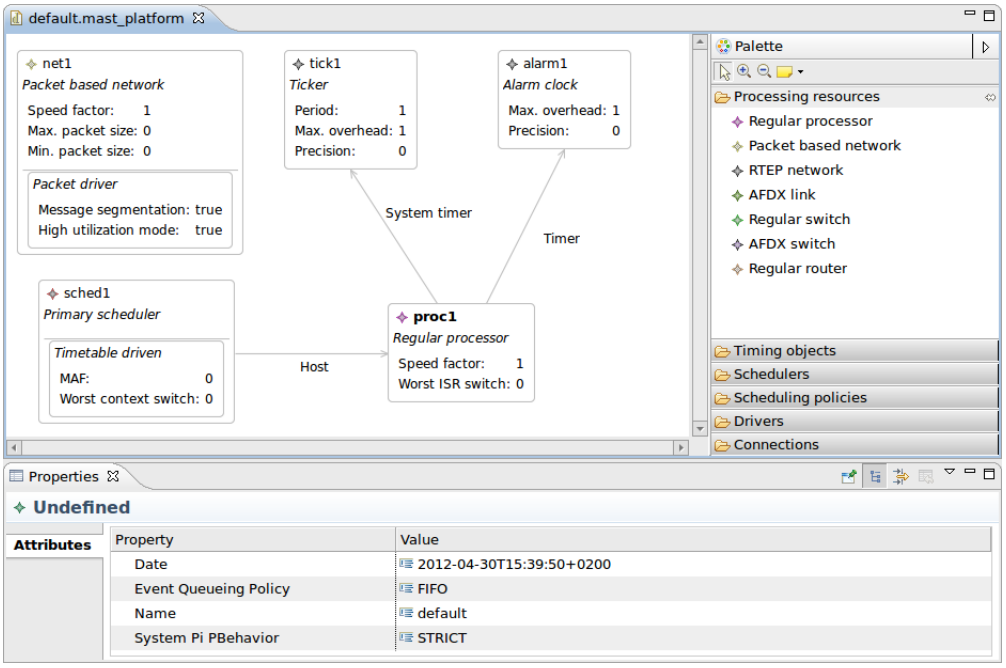

Fig. 4.5: Editor de la plataforma del modelo MAST

situar en cualquier punto del lienzo, pero los controladores de red sólo pueden añadirse dentro de una red ya existente, apareciendo como subnodo de esta.

Las conexiones entre elementos pueden establecerse, siempre que sean legítimas (estén contempladas en el modelo), directamente sobre el diagrama o utilizando la herramienta correspondiente del grupo *Connections*. Las valores de las propiedades mostrados en el diagrama son editables haciendo click sobre ellos;

para los atributos no mostrados en el diagrama es necesario acudir a la vista de propiedades, donde se listan todos los atributos del nodo seleccionado o, si no se ha seleccionado ningún nodo, del objeto raíz del modelo (la instancia de la clase `Mast_Model`).

Los únicos elemento de tercer nivel incluidos en el editor son los `Partition_Window` contenidos en el atributo `Partition_Table` de las políticas de planificación `Timetable_Driven` y `Timetable_Driven_Packet_Based`. La edición de estos objetos se realiza a través de la sección `Partition table` agregada a la vista de propiedades de estas dos clases (figura 4.6).

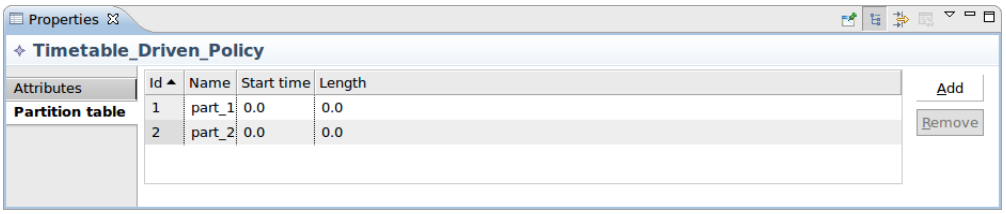

Fig. 4.6: Edición de la tabla de particiones

Los botones *Add* y *Remove* permiten respectivamente añadir y eliminar particiones a la tabla. Los elementos de la tabla pueden editarse haciendo doble click sobre el campo a editar o seleccionando la fila y pulsando *Enter* para entrar en modo de edición y el tabulador para navegar entre los diferentes campos.

4.2.4. Exportación de un modelo a XML

Para utilizar las herramientas de análisis y simulación de MAST es necesario exportar el modelo editado a XML. En el menú de exportación (*File* → *Export...*), bajo la categoría *Mast*, está el asistente *Mast model* (figura 4.7), desde donde se puede generar una versión XML del modelo MAST seleccionado.

El modelo creado por el asistente puede ser procesado directamente por las herramientas de MAST.

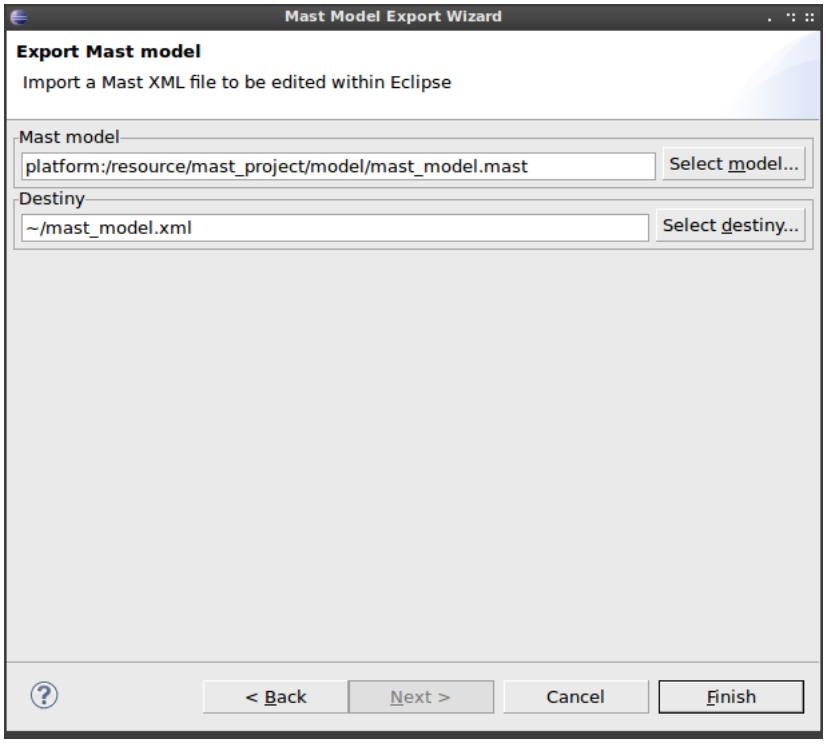

Fig. 4.7: Asistente de exportación de modelos MAST a XML

Capítulo 5

Guía de desarrollo

En este capítulo se exponen los pasos a seguir por el desarrollador interesado en ampliar los plugins con nuevos editores para diferentes particiones del modelo MAST, o bien modificar los ya existentes. No se consideran aquí características generales incompletas o mejorables (integración, asistentes, etc.), referidas en el capítulo 6.

5.1. Requisitos previos

Tal como se indicó en el capítulo 4, la plataforma utilizada durante el desarrollo del plugin ha sido *Eclipse Modeling Tools Indigo SR2*, con las siguientes dependencias para la ejecución:

- `org.antlr.runtime`, versión mayor o igual que `3.0.0` y menor que `3.1.0`.
- `org.eclipse.m2m.atl.common`
- `org.eclipse.m2m.atl.core`
- `org.eclipse.m2m.atl.core.emf`
- `org.eclipse.m2m.atl.drivers.emf4atl`
- `org.eclipse.m2m.atl.dsls`
- `org.eclipse.m2m.atl.engine.vm`

Para el desarrollo, además, es necesario el kit de herramientas de GMF. En general, el procedimiento recomendado es instalar los componentes *ATL* y *Graphical Modeling Framework Tooling* disponibles en el menú de componentes de modelado (*Help* → *Install Modeling Components*).

5.2. Obtención del código fuente

El código fuente se incluye en los archivos JAR en los que se distribuyen los plugins. Una vez instalados en Eclipse, los proyectos pueden importarse al espacio de trabajo desde la vista *Plug-ins* de la perspectiva *Plug-in development*, en la opción *Import As* → *Source Project* del menú contextual de cada plugin. El código se estructura de la siguiente manera:

`es.unican.mast` Contiene la descripción Ecore del modelo MAST, los modelos de los editores GMF, las plantillas de generación de código modificadas, las transformaciones personalizadas para crear el modelo de generación, las transformaciones ATL de importación y exportación de modelos y el esquema del formato XML de MAST (no utilizado de manera directa sino como apoyo al desarrollo). Los modelos de nuevos editores deben ubicarse en este proyecto.

`es.unican.mast.edit` Proyecto generado automáticamente por EMF con el código de edición del modelo. Sólo es necesario modificarlo si cambia el modelo Ecore (dado que la segunda versión de MAST es un trabajo en desarrollo esto puede suceder con frecuencia); en este caso, el proyecto se regeneraría con las herramientas de EMF.

`es.unican.mast.editor` Contiene clases y extensiones para integrar los diferentes editores. Los nuevos editores se deben registrar en este proyecto para ser integrados.

`es.unican.mast.editor.platform` Proyecto del editor GMF de la partición de la plataforma. La mayor parte del código es generado automáticamente y puede borrarse y regenerarse sin peligro, a excepción de las clases `TimetableDrivenFilter` y `PartitionTablePropertySection`, que implementan la sección de propiedades personalizada para las tablas de particiones.

`es.unican.mast.editor.operations` Proyecto del editor GMF de la partición de operaciones, desarrollado para probar y demostrar las características de integración de editores. El editor sólo representa la clase `Message`, y la partición de operaciones del modelo MAST no está definida de manera precisa, por lo que todo el proyecto y los modelos del editor pueden ser modificados o borrados si en el proceso de ampliación se considera conveniente.

5.3. Desarrollo de un nuevo editor

Los modelos de nuevos editores deben ubicarse en una carpeta propia, con el nombre de la partición correspondiente en minúsculas y sin espacios, bajo la carpeta `editor` del proyecto `es.unican.mast.edit`.

El proceso de desarrollo de la definición gráfica, la definición de herramientas y el mapeado del modelo es igual al de cualquier otro editor GMF (en [9] se pueden encontrar documentación y ejemplos). Sin embargo, es aconsejable adoptar las directrices seguidas en los editores ya existentes, a saber:

- Representar cada objeto como un nodo rectangular redondeado.
- Los nodos de primer nivel deben mostrar en su parte superior el nombre del objeto en negrita junto al icono de su clase.
- Todos los nodos muestran una etiqueta en letra cursiva con el nombre del tipo de objeto que representan.
- Cada nodo puede mostrar un conjunto reducido de propiedades representativas de la clase en etiquetas editables. Las propiedades se muestran verticalmente en el formato `<nombre de la propiedad>: <valor>`.
- Las asociaciones entre nodos se expresan por flechas con una etiqueta que indique el tipo asociación.
- El diagrama no muestra elementos a más de dos niveles de profundidad. Pueden utilizarse secciones de propiedades personalizadas para mostrar colecciones de objetos simples (ver sección 5.3.1).
- Las herramientas se organizan en grupos de clases relacionadas, dejando una sección para las conexiones entre elementos.

En el capítulo 3 se detallan los pasos para diseñar un editor de estas características. En concreto, puede ser útil basarse en los modelos del editor de la plataforma para mantener el mismo diseño.

Es necesario que la propiedad *Name* del elemento raíz *Canvas* de la definición gráfica sea el nombre identificativo de la partición del MAST representada en el editor (compuesta por caracteres alfanuméricos y espacios). Además, es recomendable denominar de la misma forma al elemento *Palette* de la definición de herramientas y utilizar un mismo nombre para todos los ficheros de modelos de descripción del editor (aunque no es imprescindible).

Una vez definido el editor, el modelo de generación debe ser creado utilizando las transformaciones QVTO personalizadas. Para ello, en la última página del asistente correspondiente, debe seleccionarse la opción *Use QVTO transformation* con el ruta de la transformación principal, `platform:/resource/es.unican.mast/transforms/gmfgen/Map2Gen.qvto` (figura 5.1).

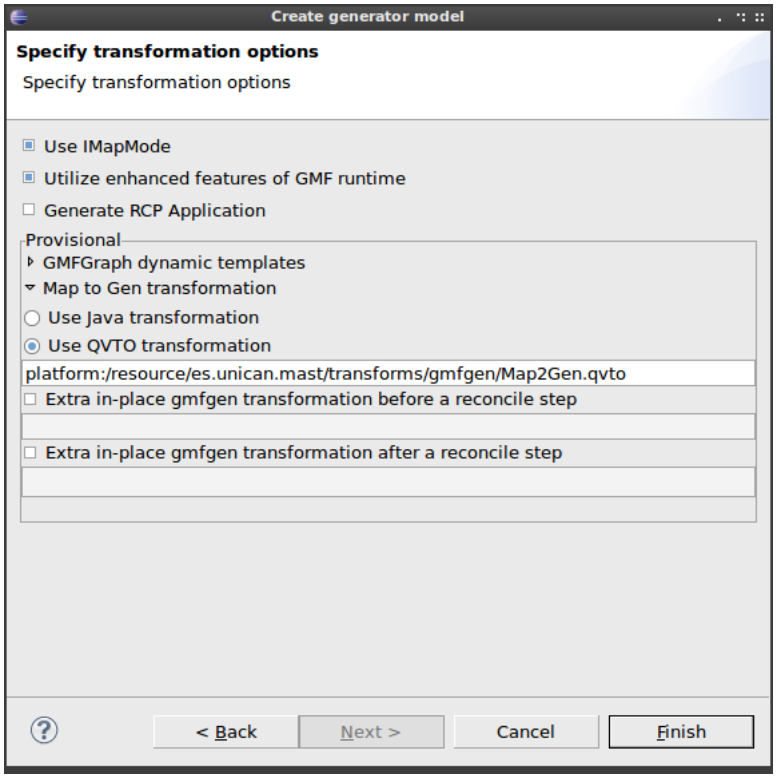

Fig. 5.1: Configuración de las transformaciones QVTO personalizadas para el modelo de generación

5.3.1. Creación de secciones de propiedades personalizadas para colecciones

Para las colecciones a mayor profundidad en la jerarquía del modelo o compuestas por un número elevado de elementos, y por lo tanto inapropiadas para ser mostradas directamente en el diagrama, existe la posibilidad de implementar secciones de propiedades personalizadas para editarlas.

Para crear una sección de propiedades personalizada son necesarios tres pasos: crear la clase que implementa la sección en sí, crear el filtro que determina las clases para las que es aplicable la sección y añadir una nueva sección de propiedades al modelo de generación.

La implementación de la sección de propiedades se realiza de manera sencilla con una clase que herede de la clase abstracta `AbstractEditableCollectionPropertySection`. Esta clase sólo debe implementar dos métodos: `getCollectionFeature` y `addBindings`. El fragmento 5.1 muestra el ejemplo de la sección de propiedades de la tabla de particiones de las subclases de `Timetable_Driven`.

El método `getCollectionFeature` debe devolver el objeto `EStructuralFeature` modelado en la sección de propiedades, obtenido desde la clase `MastPackage` generada automáticamente por EMF. En el método `addBindings` se asocian columnas de la tabla de la sección de propiedades con atributos de la clase, llamando sucesivamente a `addBinding` con el nombre de la columna y el `EAttribute` de cada atributo, obtenido de nuevo a través de `MastPackage`.

Para implementar el filtro se debe crear una subclase de `ModelClassFilter` que indique la clase de los objetos que deben presentar la nueva sección de propiedades. El fragmento 5.2 muestra el filtro para el ejemplo anterior. El método `filteredClass` devuelve el objeto `Class` de la clase de objetos que admiten la sección, en este caso `Timetable_Driven`.

Por último, debe agregarse la sección de propiedades al modelo de generación. Para ello, bajo el objeto *Property Sheet*, se añade un *Custom Property Tab* en el que se indica el nombre de la sección que verá el usuario, la clase que la implementa y un identificador único. Bajo este elemento se añade un *Custom filter* que indica la clase que implementa el filtro. En general es recomendable que las clases que implementan secciones de propiedades personalizadas estén ubicadas en el proyecto del editor correspondiente, aunque no es imprescindible.

```
package es.unican.mast.editor.platform.sheet;

import org.eclipse.emf.ecore.EStructuralFeature;

import es.unican.mast.MastPackage;
import es.unican.mast.ui.propsheet
    .AbstractEditableCollectionPropertySection;

/**
 * Property section for {@link es.unican.mast
 * .Timetable_Driven} partition window.
 *
 * @author Javier de la Dehesa
 *
 */
public class PartitionTablePropertySection extends
    AbstractEditableCollectionPropertySection {

    @Override
    protected EStructuralFeature getCollectionFeature() {
        return MastPackage.eINSTANCE
            .getTimetable_Driven_Partition_Table();
    }

    @Override
    protected void addBindings() {
        addBinding("Id",
            MastPackage.eINSTANCE
                .getPartition_Window_Partition_Id());
        addBinding("Name",
            MastPackage.eINSTANCE
                .getPartition_Window_Partition_Name());
        addBinding("Start time",
            MastPackage.eINSTANCE
                .getPartition_Window_Start_Time());
        addBinding("Length", MastPackage.eINSTANCE
                .getPartition_Window_Length());
    }
}
```

Fragmento 5.1: Implementación de la sección de propiedades de la tabla de particiones

5.3.2. Generación y mantenimiento de identificadores visuales

Para generar el editor basta con seleccionar la opción *Generate diagram code* del menú contextual del modelo de generación. El modelo tiene configuradas las plantillas de generación de código personalizadas, por lo que el código obtenido es directamente utilizable (salvo que sean necesarias clases adicionales, e. g. para secciones de propiedades personalizadas).

Es importante señalar que con la creación del modelo de generación se crea

```java
package es.unican.mast.editor.platform.filters;

import es.unican.mast.Timetable_Driven;
import es.unican.mast.ui.filters.ModelClassFilter;

/**
 * Filter for {@link es.unican.mast
 * .Timetable_Driven} scheduling policies.
 *
 * @author Javier de la Dehesa
 *
 */
public class TimetableDrivenFilter
    extends ModelClassFilter {

    @SuppressWarnings("rawtypes")
    @Override
    protected Class filteredClass() {
        return Timetable_Driven.class;
    }
}
```

Fragmento 5.2: Filtro para la sección de propiedades de la tabla de particiones

también un archivo de traza con extensión `.trace` que mantiene los identificadores visuales de cada elemento gráfico del diagrama, o `visualID`. Estos identificadores numéricos son únicos para cada tipo de objeto visual, y la validez del código del editor generado depende directamente de ellos, de manera que si uno o varios identificadores cambian suele ser necesario borrar la mayor parte del código antes de regenerarlo.

El objetivo del archivo de traza es mantener la consistencia de los identificadores, de modo que si es necesario recrear el modelo de generación (lo que es habitual en el proceso de desarrollo) se mantengan sin cambios (añadiendo nuevos identificadores si son necesarios); así, la regeneración de código no entra en conflicto con el código generado previamente.

Sin embargo, el uso de transformaciones QVTO personalizadas para el modelo de generación causa una gestión incorrecta del archivo de traza, como se ha señalado en [3], dando lugar a la creación de una traza vacía en cada creación del modelo de generación. Hay dos formas de afrontar este problema durante el proceso de desarrollo:

1. Borrar todo el código generado previamente en cada regeneración de código.

   Si no se han hecho modificaciones y no existen clases adicionales en el

proyecto del editor puede hacerse sin peligro, con el inconveniente de que los diagramas creados para versiones anteriores podrían no funcionar (aunque siempre se pueden generar nuevos diagramas a partir del modelo).

2. Obtener un archivo de traza válido creando un modelo de generación sin utilizar las transformaciones QVTO. Este archivo se puede mantener en una copia de seguridad y reemplazar al archivo inválido que se genera normalmente. Debe crearse un archivo de traza nuevo cada vez que se añadan o eliminen elementos gráficos.

Capítulo 6

Líneas futuras

En este capítulo se señalan características comunes a todos los editores que han quedado fuera del alcance de este proyecto.

6.1. Mejor integración de los editores

La interfaz ideal de los editores del modelo MAST al usuario en el entorno Eclipse es un editor divido en varias páginas, una por cada partición del modelo. La clase abstracta `MultiPageEditorPart` implementa esta funcionalidad.

Sin embargo, la dificultad de coordinar los dominios de edición (imprescindible para mantener la coherencia entre las diferentes páginas de editor), los problemas producidos por los bugs [3] y [1] y la falta de soporte en el framework de desarrollo no han permitido llevarlo a cabo.

En [15] y [25] se proponen soluciones parciales al problema que pueden servir de punto de partida para una implementación completa.

6.2. Integración de los ficheros

Una característica deseable, ligada a la anterior, es la integración de todos los ficheros de modelo y diagramas en uno solo. Aunque GMF ofrece la posibilidad de almacenar un diagrama en el mismo fichero que el modelo no es extensible a varios diagramas de editores diferentes, por lo que el enfoque más probable sería a través de técnicas comunes de compresión de archivos.

6.3. Mejora de las secciones de propiedades personalizadas

Aunque las secciones de propiedades personalizadas desarrolladas en el capítulo 3 ofrecen cierta flexibilidad, la posibilidad de manejar estructuras jerárquicas complejas y no solo lineales ampliaría notablemente su utilidad.

En esta línea, se sustituiría la actual tabla por una vista de árbol, adaptando el código de control de eventos y probablemente modificando la interfaz de programación de la clase abstracta responsable.

6.4. Integración con herramientas de MAST

En una etapa más madura del software, la integración con herramientas de MAST desde el propio entorno de Eclipse facilitaría la metodología de trabajo del analista modelador.

Para ello, se ofrecería una interfaz de configuración para indicar la ruta de los ejecutables de las herramientas de análisis y simulación, y se añadirían configuraciones de ejecución para ambas utilidades.

6.5. Visor de resultados de análisis y simulación

En paralelo con el editor gráfico del modelo MAST, es necesario un visor intuitivo de los resultados del procesado de modelos. Dado que el visor ofrecería una interfaz de solo lectura, sería necesario analizar la adecuación de GMF al problema y, de no ser satisfactoria, buscar frameworks alternativos bajo la plataforma Eclipse.

Capítulo 7

Conclusiones

En este trabajo se ha planteado una respuesta a la necesidad de un editor gráfico para el modelo MAST, usando GMF como framework de desarrollo. Antes de dar una evaluación final de la viabilidad del editor completo, repasaremos las fortalezas y debilidades del framework.

7.1. Fortalezas de GMF

7.1.1. Potencia

GMF es capaz de generar un editor sencillo plenamente funcional en cuestión de minutos, incluso sin conocimientos previos de la plataforma. Esto lo sitúa muy por delante de la mayoría de alternativas, que por lo general requieren un conocimiento relativamente amplio del framework para obtener un resultado similar.

Por otro lado, la cantidad de opciones disponibles es abrumadora: configuraciones gráficas virtualmente ilimitadas, capacidad para definir todo tipo de menús y herramientas, mapeados totalmente arbitrarios con soporte para restricciones OCL, etc. Las posibilidades del editor final son realmente extensas.

7.1.2. Flexibilidad

Aunque se trate de una funcionalidad avanzada, la posibilidad de personalizar las transformaciones del modelo de generación y las plantillas de generación

de código permiten configurar hasta el más mínimo detalle del editor de un modo reutilizable. Es relativamente sencillo personalizar los comportamientos más básicos del editor de acuerdo a las necesidades del proyecto.

7.2. Debilidades de GMF

7.2.1. Madurez

Así como EMF es un proyecto de una madurez y solidez incuestionable, no puede decirse lo mismo de GMF. No es difícil encontrar inconsistencias más o menos irrelevantes, como pueden algunas asignaciones de valores predeterminados y otros pequeños errores puntuales (sin ir más lejos, algunos de los asistentes que incluye el framework producen ocasionalmente resultados incorrectos). A lo largo de este trabajo se ha explotado sólo un pequeño conjunto de todas las opciones que ofrece GMF, y sin embargo han sido varios los bugs que han dificultado o impedido un desarrollo correcto y completo [1][2][3].

Otra consecuencia directa de la falta de madurez es la escasez de documentación adecuada. Si bien es cierto que en la web del proyecto [9] existen varias guías de iniciación y ejemplos, no se cubren la mayor parte de las características avanzadas del framework, siendo necesario con frecuencia recurrir a fuentes de información adicionales, como los foros de la comunidad de Eclipse, ejemplos publicados por terceros o incluso fragmentos del mismo código fuente (con frecuencia también pobremente documentado). Tampoco existe bibliografía al respecto, a excepción de [12], que, si bien no es un tratado o manual específico de GMF, recoge una referencia completa del framework.

7.2.2. Complejidad

Como contrapartida de la potencia y flexibilidad anteriormente señaladas, el uso avanzado de GMF puede resultar extremadamente complicado. Esto se debe en parte a la ya mencionada falta de documentación, pero también a la falta de apoyo al desarrollo en ciertos aspectos, especialmente en editores de complejidad elevada (con gran número de elementos). Por ejemplo, no es posible organizar los nodos, conexiones y etiquetas de la definición gráfica en grupos, siendo imprescindible una estricta convención en la nomenclatura para mitigar las posibilidades de cometer fallos en las asociaciones; tampoco es posible definir

un objeto gráfico base común a partir del cual definir otras figuras (e. g. la forma básica de los nodos, los colores, etc.).

Del mismo modo, existen funcionalidades que, si bien son implementables mediante técnicas avanzadas, resultaría apropiado que fueras soportadas de antemano por el framework. Un ejemplo claro es la integración de varios editores GMF en un único editor con pestañas, que resulta ideal no solo para MAST sino para cualquier modelo relativamente complejo.

7.3. Evaluación de la viabilidad del editor

Es posible implementar un editor gráfico completo para MAST en GMF, tal y como ha quedado patente en este trabajo. Siguiendo iterativamente los pasos descritos en el capítulo 5 se obtendrían los editores para cada una de las particiones restantes del modelo.

Sin embargo, la funcionalidad del editor es limitada. El editor actualmente existente para la primera versión de MAST es evidentemente más consistente (en términos de madurez) que el trabajo desarrollado aquí, y ofrece una integración de las particiones del modelo totalmente transparente para el usuario. Utilizar este editor como base para el editor de MAST 2 implicaría adaptar las particiones existentes a la nueva versión del modelo, incluir las nuevas clases y añadir el soporte para el formato XML; si bien no es un trabajo trivial, la dificultad y el resultado obtenido es fácilmente predecible.

Seguir con el trabajo aquí expuesto implicaría, aparte de la implementación de los editores faltantes, desarrollos en las líneas indicadas en el capítulo 6. Este esfuerzo, probablemente más complejo e impredecible, se vería recompensado, en el mejor de los casos, con un entorno integrado de modelado, análisis y simulación del modelo MAST sobre Eclipse.

La decisión del camino a seguir vendrá tomada por las necesidades y los recursos del grupo CTR de la Universidad de Cantabria, último responsable y mantenedor del modelo MAST y sus herramientas asociadas.

Bibliografía

[1] Eclipse Bugs. *Bug 244517 - Trace model does not manage diagram visual IDs*. 2012. URL: https://bugs.eclipse.org/bugs/show_bug.cgi?id=244517.

[2] Eclipse Bugs. *Bug 354467 - "Create undefined" in create context menu*. 2012. URL: https://bugs.eclipse.org/bugs/show_bug.cgi?id=354467.

[3] Eclipse Bugs. *Bug 369562 - [QVTO-Bridge] Use gmf trace model to generate correct visual ids*. 2012. URL: https://bugs.eclipse.org/bugs/show_bug.cgi?id=369562.

[4] Grupo de Computación y Tiempo Real de la Universidad de Cantabria. *MAST home page*. 2012. URL: http://mast.unican.es.

[5] César Cuevas y col. «Estrategias MDE en entornos de desarrollo de sistemas de tiempo real». En: *Actas de las XV Jornadas de Tiempo Real*. 2012.

[6] José M. Drake y col. «MAST 2: Nueva revisión del modelo de tiempo real». En: *Actas de las XIII Jornadas de Tiempo Real*. 2010, págs. 181 -190.

[7] The Eclipse Foundation. *Eclipse ATL*. 2012. URL: http://www.eclipse.org/atl/.

[8] The Eclipse Foundation. *Eclipse Downloads*. 2012. URL: http://eclipse.org/downloads/.

[9] The Eclipse Foundation. *Graphical Modeling Framework - Eclipsepedia*. 2012. URL: http://wiki.eclipse.org/GMF.

[10] The Eclipse Foundation. *Naming Conventions - Eclipsepedia*. 2012. URL: http://wiki.eclipse.org/Naming_Conventions.

[11] The Eclipse Foundation. *Where did Eclipse come from?* 2004. URL: http://wiki.eclipse.org/FAQ_Where_did_Eclipse_come_from%3F.

[12] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, 2009.

[13] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. 2007. URL: http://www.omg.org/cgi-bin/doc?ptc/2007-07-072.

[14] Michael G. Harbour y col. «Modeling distributed real-time systems with MAST 2». En: *Journal of Systems Architecture* (2012). DOI: 10.1016/j.sysarc.2012.02.001.

[15] Jan Köhnlein. *How to share an editing domain between several GMF editors*. 2009. URL: http://code.google.com/p/gmftools/wiki/SharedEditingDomain.

[16] Herman Kopetz. «Real-Time Communication». En: *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2011. Cap. 7.

[17] Andrew Krause. *Foundations of GTK+ Development*. Apress, 2007.

[18] Andrew Krause. *OSGi and Equinox: Creating Highly Modular Java Systems*. Addison-Wesley, 2010.

[19] Jane W. S. Liu. «Typical real-time applications». En: *Real-time systems*. Prentice Hall, 2000. Cap. 1.

[20] José M. Martínez y Michael González Harbour. «A Fixed-Priority Real Time Communication Protocol over Standard Ethernet». En: *Reliable Software Technologies — Ada-Europe 2005*. 2012, págs. 180 -195.

[21] Aurélien Pupier. *Customize your GMF editor by customizing templates*. 2010. URL: http://www.bonitasoft.org/blog/eclipse/customize-your-gmf-editor-by-customizing-templates/.

[22] Trygve M. H. Reenskaug. «The Model-View-Controller (MVC) Its Past and Present». En: JavaZONE Conference. 2003.

[23] John A. Stankovic. «Misconceptions about real-time computing: a serious problem for next-generation systems». En: *Computer* 21.10 (1988), págs. 10-19. ISSN: 0018-9162. DOI: 10.1109/2.7053.

[24] S. Tucker Taft y col. *Ada 2005 Reference Manual. Language and Standard Libraries*. Springer, 2006.

[25] Volker Wegert y Alex Shatalin. *Integrating EMF and GMF Generated Editors*. 2008. URL: http://www.eclipse.org/articles/Article-Integrating-EMF-GMF-Editors/index.html.

Glosario

AFDX *Avionics Full-DupleX switched ethernet*. Protocolo para comunicaciones en tiempo real basado en Ethernet. 8, 9

API *Application Programming Interface*. Interfaz de programación ofrecida por un componente software para su uso por parte de un tercero. 4

ATL Lenguaje estándar de transformación de modelos. 30, 33, 42, *véase* M2M

bug Error genérico en un software que produce un resultado incorrecto o inesperado. 49, 52

CamelCase Estilo de escritura en el que se eliminan los espacios o separadores entre las palabras, que se escriben con la inicial en mayúscula y el resto en minúscula para mantener la legibilidad. 14

CTR Computación y Tiempo Real. Grupo de investigación de la Universidad de Cantabria dedicado al área de los sistemas de tiempo real. Responsable del proyecto MAST. 1, 6, 12, 30, 53, *véase* MAST

definición de herramientas Descripción de la paleta de herramientas de un editor GMF. Almacenada en un archivo `gmftool`. 5, 12, 13, 18–20, 24, 43, 44, *véase* GMF

definición gráfica Descripción de los elementos visuales de un editor GMF. Almacenada en un archivo `gmfgraph`. 5, 12, 13, 17, 19, 20, 24, 43, 44, 52, *véase* GMF

Eclipse Plataforma de código abierto para la construcción de aplicaciones de escritorio multiplataforma. Utilizado indistintamente para referirse al entorno de desarrollo integrado construido sobre la misma. 2–5, 11, 12, 21, 22, 28, 30, 33, 34, 42, 49, 50, 52, 53

Ecore Metamodelo incluido dentro de EMF para describir modelos de datos genéricos de una modo independiente del lenguaje. 4, 5, 12, 13, 19, 20, 23, 29, 42, *véase* EMF

EDF *Earliest Deadline First*. Política de planificación en la que los recursos se planifican en orden de plazo de ejecución más corto. 9

EMF *Eclipse modeling framework*. Framework de modelado para aplicaciones basadas en modelos de datos estructurados. 4, 6, 18, 23, 28–30, 34, 42, 45, 52, *véase* Eclipse y framework

Equinox Implementación del estándar OSGi usada por la plataforma Eclipse. 3, *véase* OSGi y Eclipse

extensión Funcionalidad ofrecida por un paquete que se integra al sistema a través de un punto de extensión (e.g. en un procesador de textos, la extensión "Times new roman" podría integrarse un punto de extensión "Fuente"). 3, 5, 22, 23, 26–28, 30, 42, *véase* Eclipse y punto de extensión

FIFO *First In, First Out*. Característica de una estructura de datos de lectura y escritrua por la cual los elementos son leídos o extraídos en el mismo orden en el que entraron a la estructura. 8

framework Conjunto de especificaciones, librerías y herramientas software que proporcionan apoyo al desarrollo de otros proyectos. 2, 4, 9, 12, 27, 49–53

GMF *Graphical Modeling Framework*. Framework de desarrollo de editores gráficos basados en la plataforma Eclipse. 2, 4–7, 9, 11–14, 16, 21, 24, 26, 30, 41–43, 49–53, *véase* Eclipse y framework

holgura En sistemas de tiempo real, porcentaje en el que un parámetro puede ser incrementado o reducido manteniendo la planificabilidad del sistema. 6, *véase* sistema de tiempo real y planificabilidad

M2M *Model to model*. Conjunto de técnicas que permiten convertir una instancia de un metamodelo a una de otro metamodelo. 6, 29

M2T *Model to text*. Conjunto de técnicas que permiten convertir un modelo en código fuente. 6

manifiesto Información asociada a un paquete en la que se incluyen datos comunes (nombre, versión, etc.) y los requisitos para poder ser utilizado, principalmente los paquetes de los que depende y, en su caso, la versión. 3, *véase* paquete

mapeado del modelo Asociación de los elementos de un modelo Ecore con componentes de una definición gráfica y una definición de herramientas. Almacenado en un archivo `gmfmap`. 5, 12, 13, 17, 19–21, 24, 43, *véase* GMF, Ecore, definición gráfica y definición de herramientas

MAST *Modeling and Analysis Suite for real-Time applications*. Modelo que describe el comportamiento temporal de un sistema de tiempo real diseñado para ser analizado mediante técnicas de análisis de planificabilidad. 1, 2, 6, 7, 11–13, 28–30, 33, 34, 36, 38, 41, 42, 44, 49–51, 53, *véase* planificabilidad

modelo de generación Archivo creado a partir de un modelo Ecore, una definición gráfica, una definición de herramientas y un mapeado del modelo con la información necesaria para generar el código de un editor GMF. Almacenado en un archivo `gmfgen`. III, V, 5, 6, 12, 21, 23, 24, 26, 27, 42, 44–48, 51, *véase* GMF, Ecore, definición gráfica, definición de herramientas y mapeado del modelo

MVC Modelo-vista-controlador. Patrón de diseño software que divide el desarrollo en tres particiones independientes: modelo, que describe el modelo de datos subyacente, vista, que define de los elementos visuales con los que interacciona el usuario, y controlador, que rige la relación entre los dos anteriores e implementa la lógica de negocio. 6

OCL *Object Constraint Language*. Lenguaje de definición de restricciones sobre modelos de objetos. 19, 51

OSGi *Open Services Gateway initiative*. Estándar para la construcción de aplicaciones modulares en lenguaje Java. 3

paquete Conjunto de artefactos software encapsulados que ofrecen una funcionalidad determinada y potencialmente un plugin. 3, 33, *véase* plugin y Eclipse

planificabilidad Cualidad de un sistema de tiempo real por la que garantiza, mediante análisis formal, el cumplimiento de sus restricciones temporales. 6

plugin Paquete que implementa una o más extensiones y/o puntos de extensión. 3, 5, 21, 24, 27, 28, 30, 33, 34, 41, 42, *véase* extensión, punto de extensión y Eclipse

punto de extensión Interfaz ofrecida por una extensión para integrar un nuevo tipo de extensiones (e.g. un procesador de textos podría ofrecer un punto de extensión "Fuente"). 3, *véase* Eclipse y extensión

QVTO Lenguaje estándar de transformación de modelos. 24, 44, 47, 48, *véase* M2M

RTEP *Real-Time Ethernet Protocol*. Protocolo para comunicaciones en tiempo real basado en Ethernet. 8, 9

sistema de tiempo real Aquel en el que la corrección del cómputo no depende solo de su corrección lógica sino también del tiempo en el que se produce. 1, 7, 24, 28

SWT *Standard Widget Toolkit*. Librería de componentes visuales para el diseño de interfaces gráficas en Java, y en particular para el desarrollo de extensiones para Eclipse. 22, *véase* Eclipse y extensión

XMI *XML Metadada Interchange*. Lenguaje estándar de intercambio de metadatos basado en XML. 29, 30, *véase* XML

XML *eXtensible Markup Language*. Lenguaje extensible basado en etiquetas, diseñado para transmitir información de manera fácilmente procesable tanto por humanos como por máquinas.. 1, 27–30, 34, 38, 42, 53

Xpand Lenguage de generación de código basado en los modelos EMF. 27, *véase* EMF y M2T

XSTL *eXtensible Stylesheet Language Transformations*. Variante de XML usada para la transformación de documentos XML. 29, *véase* XML y M2M