Programa Oficial de Postgrado en Ciencias, Tecnología y Computación

Máster en Computación

Facultad de Ciencias - Universidad de Cantabria

Integración de las herramientas de análisis MAST en un entorno de desarrollo dirigido por modelos y basado en Eclipse

Melitón Pablo Mangué Mañana

mmm68@alumnos.unican.es

Directores: Patricia López Martínez y César Cuevas Cuesta

Grupo de Computadores y Tiempo Real

Santander, Octubre de 2013

Curso 2012/2013

Agradecimientos:

- En primer lugar, agradecer a mi directora Patricia López Martínez por toda la ayuda y por la rapidez con la que me rescataba y revisaba mis avances.

- En segundo lugar a César Cuevas Cuesta, por estar siempre dispuesto a interrumpir lo suyo para atender a mis dudas.

A mi madre Pilar.

Resumen

MAST (*Modelling and Analysis Suite for Real-Time Applications*) es un entorno para el modelado y análisis de sistemas de tiempo real, desarrollado por el grupo de Computadores y Tiempo Real (CTR) de la Universidad de Cantabria. MAST proporciona una metodología de modelado para la formulación del modelo de tiempo real de un sistema, así como un conjunto de herramientas que pueden aplicar diferentes tipos de análisis sobre los modelos.

CTR está actualmente está trabajando en una nueva versión del entorno MAST, que trata de adaptarlo a una visión más orientada a MDE. Los metamodelos de la nueva versión MAST ya han sido definidos, sin embargo, las herramientas todavía no han sido actualizadas. En este caso, para usar estas herramientas es necesario adaptarlas para trabajar con modelos MAST 2.0.

CTR también trabaja actualmente en la aplicación de MDE al desarrollo de sistemas de tiempo real, en particular mediante el desarrollo de una metodología para construir entornos de diseño de tiempo real plenamente dirigidos por modelos. Al aplicar MDE al diseño de tiempo real se consigue un incremento de la facilidad de uso del entorno para el diseñador de aplicaciones. Por otro lado, también se permite que el desarrollador de herramientas de análisis y diseño de planificabilidad disponga de modelos mucho más simplificados, al elevarse el nivel de abstracción.

El entorno diseñado para validar la metodología propuesta se está implementando sobre la plataforma Eclipse porque su *framework* de modelado soporta de forma natural la disciplina MDE. Dicho entorno, como parte del proceso de desarrollo de una aplicación de tiempo real, deberá integrar las herramientas MAST – en Ada –, pero con capacidad de lanzamiento desde el propio entorno y mostrando los resultados a través de sus propios recursos, los que proporciona Eclipse en este caso. Este trabajo aborda dicho proceso de integración desde un punto de vista genérico, es decir, aplicable a cualquier herramienta externa que se quiera integrar en el entorno. El proceso se particulariza para el caso de las herramientas de análisis MAST.

Siglas

API ..... Application Programming Interface

EMF ..... Eclipse Modeling Framework

MAST ..... Modeling and Analysis Suite for Real-Time Systems

MDA ..... Model Driven Architecture

MD(S)D..... Model Driven (Software) Development

MD(S)E..... Model Driven (Software) Engineering

OCL ..... Object Constraint Language

OMG ..... Object Management Group

OO ..... Object Oriented

RMA ..... Rate Monotonic Analysis

SAX ..... Simple API for XML

UML ..... Unified Modeling Language

URI ..... Uniform Resource Identifier

URL ..... Uniform Resource Locator

W3C ..... World Wide Web Consortium

XMI ..... XML Metadata Interchange

XML ..... eXtensible Markup Language

Contenido

Índice de figuras:

1 Introducción

En este capítulo se plantean los objetivos del trabajo realizado. Antes de dar a conocer en detalle dicho objetivos, se realiza una pequeña introducción a los conceptos manejados en el trabajo.

1.1 Ingeniería Dirigida por Modelos

La Ingeniería Dirigida por Modelos (*Model-Driven Engineering*, MDE) [1] es una disciplina dentro de la Ingeniería del Software que hace hincapié en la creación de software a partir del uso sistemático de modelos, los cuales desempeñan el papel principal de todo el proceso de desarrollo y ciclo de vida del software, frente a las alternativas tradicionales basadas en lenguajes de programación, plataformas de objetos y componentes software. MDE persigue elevar el nivel de abstracción en el desarrollo de software, convirtiendo a los modelos y a las transformaciones de modelos en los principales artefactos de todas las fases del proceso de desarrollo (captura y gestión de requisitos, diseño, análisis, implementación, despliegue, configuración, mantenimiento, evolución, etc.). Con ello se busca aumentar la productividad, reducir los errores de programación y facilitar la adaptación del software a cambios futuros y/o nuevos requisitos, lo que conduce al objetivo final de disminuir el coste y mejorar la calidad de las inversiones en software.

Un modelo es una representación abstracta de un dominio, sistema o entidad del mundo real que captura sus elementos importantes y las relaciones entre ellos. Los elementos de representación utilizados en el modelo son dependientes del formalismo de modelado utilizado. Por ejemplo, en UML se representa el problema con clases y relaciones y se utilizan elementos gráficos para mostrarlos.

En MDE, cada modelo ha de ser conforme a un metamodelo. Un metamodelo describe la estructura y características de los modelos, restringiendo el tipo de elementos que pueden aparecer en los modelos y las relaciones que se pueden establecer entre ellos. En definitiva, las características de cada elemento estructural del modelo se describen en el metamodelo.

Las principales soluciones que aporta MDE son:

- Separar los aspectos del dominio de los aspectos de la tecnología.
- Facilitar la reutilización de conocimiento mediante su registro en los modelos y en las transformaciones.
- Automatizar partes significativas del proceso de desarrollo. El proceso de transformar una especificación software en un programa ejecutable será automático, por tanto el código fuente de las aplicaciones se debería generar a partir de los modelos en un proceso de transformación. En esencia, se puede definir el modelo y generar código desde él mismo, construyendo así el esqueleto de la aplicación – y, a veces toda la aplicación – basada en este modelo.

1.2 Sistemas de Tiempo Real

1.2.1 Concepto de sistema de tiempo real

Un sistema de tiempo real es una combinación de uno o varios computadores, dispositivos de hardware de I/O y software de propósito especial, en el que:

- Hay una fuerte interacción con el entorno.
- Ese entorno cambia con el tiempo.
- El sistema controla o reacciona de forma simultánea a diferentes aspectos del entorno.

Debido a esta fuerte interacción con el entorno, el funcionamiento correcto de las aplicaciones de tiempo real no sólo depende de los resultados del cálculo, sino también del instante en el que se generan esos resultados. Por tanto, para poder asegurar que un sistema de tiempo real es correcto, en necesario imponer una serie de requisitos temporales a las respuestas que implementa el sistema. Estos requisitos forman parte de la especificación del sistema, y en consecuencia, el sistema deberá ser implementado de modo que se garantice su cumplimiento.

De acuerdo con la severidad de los requisitos temporales, los sistemas de tiempo real pueden clasificarse en:

- Sistemas de tiempo real estricto: el incumplimiento de un requisito es un fallo irrecuperable y de consecuencias graves, por lo que nunca debe producirse.
- Sistemas de tiempo real laxo: el incumplimiento de un requisito es un fallo que influye en la calidad de servicio del sistema pero no tiene consecuencias críticas, por lo tanto es suficiente con que los requisitos se cumplan en promedio.

Para asegurar el cumplimiento de los requisitos temporales impuestos, el proceso de desarrollo de un sistema de tiempo real debe incluir una fase de análisis de planificabilidad, a través de la cual se verifique el cumplimiento de dichos requisitos incluso en el peor caso. Para poder aplicar herramientas de análisis de planificabilidad es necesario contar con un modelo de tiempo real del sistema, esto es, un modelo que describa el comportamiento temporal de las diferentes respuestas del sistema a los eventos del entorno. Los requisitos impuestos en la especificación del sistema se plasman también en dicho modelo, y de ese modo, las herramientas que lo procesan pueden evaluar su cumplimiento.

1.2.2 Entorno MAST

MAST (*Modelling and Analysis Suite for Real-Time Applications*)[1] [2][3] es un entorno para el modelado y análisis de sistemas de tiempo real, desarrollado por el grupo de Computadores y Tiempo Real (CTR) de la Universidad de Cantabria. MAST proporciona una metodología de modelado para la formulación del modelo de tiempo real de un sistema, así como un conjunto de herramientas que pueden aplicar diferentes tipos de análisis sobre los modelos.

La Fig. 1-1 muestra como el entorno MAST se centra en un modelo de datos compuesto a su vez por tres modelos:

[1] http://mast.unican.es

- Modelo de descripción del sistema: describe el comportamiento temporal del sistema como un conjunto de transacciones (secuencias de actividades relacionadas por flujo de control) ejecutadas en respuesta a eventos externos o procedentes del reloj del sistema.
- Modelo de resultados: describe los resultados obtenidos después de aplicar el análisis correspondiente.
- Modelo de trazas: describe las trazas de ejecución de la simulación del comportamiento temporal del sistema.

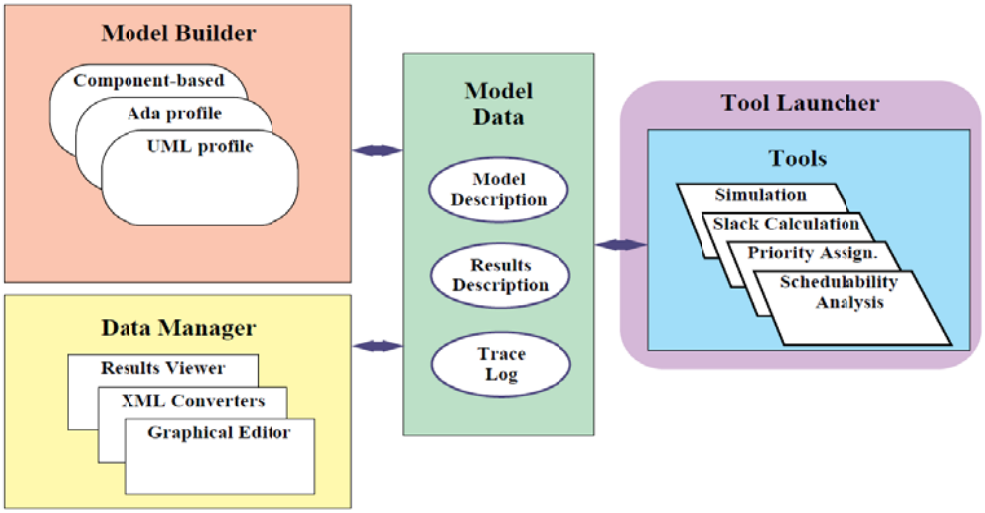

Figura 1-1 Estructura del entorno MAST

La Fig. 1-1 también muestra el conjunto de herramientas que, tomando como entrada el modelo de descripción del sistema, produce los modelos de resultados y de trazas. MAST ofrece cuatro tipos de herramientas:

- Herramientas de análisis de planificabilidad, que, aplicando técnicas analíticas de peor caso, permiten verificar el cumplimiento de requisitos de tiempo real estricto.
- Herramienta de simulación, que, aplicando técnicas basadas en simulación, permite verificar el cumplimiento de requisitos de tiempo real laxo.
- Herramientas de cálculo de holguras, que permiten evaluar cómo de lejos está un sistema de ser planificable (o cómo de cerca de dejar de serlo).
- Herramientas de asignación de parámetros de planificación, que permiten calcular los parámetros de planificación – prioridades y techos de recursos compartidos - que hacen a un sistema planificable.

Las características más importantes de MAST son:

- Utiliza un rico modelo orientado a eventos en el que se pueden establecer diferentes patrones de dependencia entre las respuestas del sistema.
- Utiliza técnicas de análisis basadas en *offsets* que mejoran en gran medida los resultados obtenidos del análisis por técnicas tradicionales como RMA (*Rate Monotonic Analysis*).
- El conjunto de herramientas es de código abierto y totalmente extensible. Esto significa que otros equipos pueden proporcionar mejoras y/o desarrollar nuevas herramientas de análisis.

El entorno MAST se encuentra actualmente en pleno proceso de evolución hacia su versión 2.0, tratando de adaptarlo a una visión más orientada a MDE. Los modelos de datos de la nueva versión ya han sido definidos, dando lugar a los metamodelos MAST 2.0 [4], MAST Results 2.0 y MAST Traces 2.0. Estos nuevos metamodelos incorporan nuevos elementos de modelado y redefinen mejor algunos de los anteriores [5]. Sin embargo las herramientas, excepto el simulador, todavía no han sido actualizadas. Esto hace que la mayoría de ellas sigan admitiendo únicamente como entrada modelos acordes a la versión MAST 1.4.

1.3 Aplicación de MDE al desarrollo de Sistemas de Tiempo Real

1.3.1 Beneficios de la aplicación de MDE

En esta sección se analizan los beneficios que aporta la aplicación de MDE al desarrollo de sistemas de tiempo real. Primero se presentan los objetivos que guían el diseño de cualquier entorno de desarrollo de software acorde a MDE:

- Integrar de forma natural los diversos aspectos y dominios que conciernen al diseño de un sistema software.
- Hacer sencillo su uso a los usuarios finales (los diseñadores de sistemas software), ofreciendo una estrategia estandarizada e integrada de acceso a los diferentes dominios (vistas), información (modelos) y procesos (herramientas).
- Facilitar el mantenimiento y la extensión del entorno a los diseñadores de dominios, modelos y procesos, ofreciendo en el propio entorno especificaciones, modelos y herramientas específicos a su tarea.
- Fomentar la modularidad, estandarización y reusabilidad de los elementos del entorno, así como la portabilidad e inter-operatividad del entorno.

Además de estos beneficios genéricos, en [6] se analizan los beneficios que se obtienen al aplicar un enfoque MDE al diseño de sistemas de tiempo real, entre otros:

- Incrementar la facilidad de uso del entorno al permitir la definición de modelos orientados a los diferentes paradigmas de diseño de las aplicaciones de tiempo real.
- Facilitar el desarrollo e integración de nuevas herramientas de análisis y/o diseño con el nivel de abstracción que se necesita. Las herramientas MDE automatizan las transformaciones entre los modelos y el mantenimiento de la coherencia entre ellos.

Lo que se consigue al aplicar MDE al desarrollo de sistemas de tiempo real es el incremento de la facilidad de uso del entorno para el diseñador de aplicaciones de tiempo real, ya que, en función del paradigma que utilice y de la fase del proceso de diseño que esté llevando a cabo, el entorno le ofrece una vista especializada del sistema, en la que dispone de modelos que utilizan únicamente los elementos que corresponden a los entes que le son familiares al diseñador. Por ejemplo, si se aplica un paradigma de desarrollo de sistemas embebidos a bajo nivel de abstracción, se pueden ofrecer elementos de modelado de bajo nivel como tareas, *mutex*, operaciones, etc. En cambio, cuando se aplica un paradigma de diseño modular (orientación a objetos, componentes, etc.) los elementos de modelado deberían ser componentes, objetos (activos, pasivos y protegidos), servicios de *middleware*, planes de despliegue, etc. Asimismo, un entorno MDE facilita la labor del desarrollador de herramientas de análisis y diseño de planificabilidad, las cuales constituyen el núcleo fundamental de un entorno de diseño de

sistemas de tiempo real. El desarrollador requiere en este caso modelos mucho más simplificados, con sólo aquellos elementos y asociaciones que corresponden a las abstracciones procesadas por las herramientas que se diseñan. Por ejemplo:

- En el caso de herramientas de análisis de planificabilidad, el modelo se formularía de acuerdo a un diseño reactivo que describa las respuestas que se ejecutan, el modelo de carga (patrón de activación de las respuestas) y los requisititos temporales, así como un modelo de recursos que describa los retrasos por suspensión y bloqueo que ocurren como consecuencia del acceso a los mismos en exclusión mutua.
- En el caso de las herramientas de análisis de planificabilidad basado en reserva de recursos, se manejarían conceptos como contratos de servicio, niveles de concurrencia, modelos de carga, requisitos temporales, etc.
- Por último, en el caso de análisis mediante simulación, el modelo ha de ser reformulado para que se incremente su eficiencia de ejecución e incorpore los observadores necesarios para que se capture la información de la ejecución que requiere el diseñador.

Por tanto, el empleo de MDE incrementa la facilidad de uso del entorno y el desarrollo de herramientas, en base a que cada diseñador de aplicaciones y cada desarrollador de herramientas pueden disponer de modelos adecuados a su tarea específica. Además, gracias a la gestión y transformación de los modelos se consigue que cada herramienta integrada en el entorno reciba únicamente la información para la que ha sido diseñada y que por tanto, sabe cómo procesar.

Existe un beneficio adicional debido a que MDE proporciona a bajo coste soporte para el conjunto de operaciones de gestión de información que son independientes de la naturaleza de los modelos procesados, lo cual simplifica enormemente el desarrollo de herramientas específicas que necesiten hacer uso de éstas operaciones. Se evita así que estos gestores de información formen parte de las herramientas específicas.

La vista de diseño de tiempo real de una aplicación se integrará coherentemente en un entorno de desarrollo más general, el cual soporta las restantes fases del diseño del sistema. Todas las fases se benefician de las ventajas de MDE expuestas.

1.3.2 Entorno RT-MDE

Una de las líneas de investigación que actualmente sigue el grupo de CTR es la aplicación de MDE al diseño de sistemas de tiempo real, la cual comprende la especificación de una metodología (RT-MDE) para la construcción de entornos de desarrollo basados en MDE y destinados a tales aplicaciones.

En consecuencia, este trabajo se enmarca en un entorno que sigue dicha metodología. Un entorno RT-MDE integra:

- Los modelos mediante los que se formula la información acerca del sistema bajo desarrollo.
- Las herramientas, tanto genéricas como específicas, mediante las que se realizan los procesos de desarrollo soportados por el entorno.
- Los mecanismos y recursos (textuales, tabulares y gráficos) para la interacción con el operador.

Para la validación de la metodología, se ha implementado el entorno sobre la plataforma Eclipse porque su *framework* de modelado soporta de forma natural la disciplina MDE, la

infraestructura del *workbench* de Eclipse posibilita una implementación inmediata y estandarizada del entorno y su arquitectura modular (basada en *plug-ins*) hace que el mecanismo de integración de nuevas funcionalidades sea robusto y confiable.

La plataforma elegida (Eclipse) y el dominio y metodología asociada (en nuestro caso tiempo real y MAST, respectivamente) son ortogonales. Se tiene entonces una implementación de RT-MDE sobre Eclipse orientada a dar soporte al desarrollo de sistemas de tiempo real utilizando MAST. La Fig. 1-2 muestra esta visión, ilustrando que Eclipse ofrece la infraestructura sobre la que implementar el entorno mientras que éste se orienta para dar servicio al diseño de tiempo real aplicando MAST.

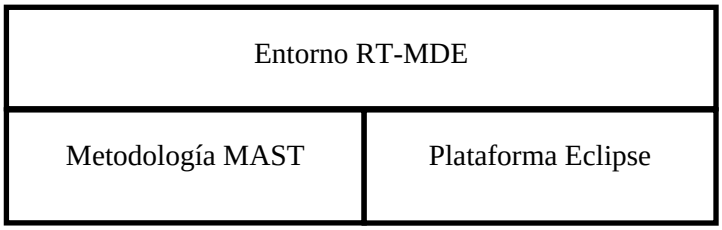

Figura 1-2. Estructura del entorno RT-MDE sobre Eclipse y orientado a MAST

Soporte MDE ofrecido por Eclipse

La plataforma Eclipse proporciona soporte MDE a través de su proyecto de primer nivel *Eclipse Modeling Project* (EMP)[1] y en concreto su subproyecto *Eclipse Modeling Framework* (EMF)[2] [7]. El núcleo de EMF está constituido por el lenguaje de metamodelado Ecore, cuya principal ventaja es su simplicidad (mucho más reducido que el MOF definido por OMG) y que cubre básicamente los aspectos estructurales de los metamodelos y sólo residualmente algo relativo al modelado del comportamiento. Entre otras características, EMF proporciona serialización XMI tanto para los meta-modelos formulados mediante Ecore como para los modelos conformes a ellos, así como editores genéricos para su manipulación y la posibilidad de generar automáticamente una representación de un metamodelo codificada en Java. Sin embargo, lo más importante de todo es que EMF supone la base para la interoperabilidad con todo el resto de subproyectos y herramientas de EMP.

Una consecuencia de utilizar EMF como infraestructura de soporte MDE es que los procesos de desarrollo se han de basar en las formulaciones en Ecore de los metamodelos implicados (en nuestro caso, MAST 2.0 y MAST Results 2.0), trabajando con modelos conformes a ellos, y por tanto su representación canónica es la serialización XMI ofrecida por EMF.

La Fig. 1-3 muestra los elementos principales de Ecore, que se describen brevemente a continuación:

- **EClass:** representa una clase que contiene atributos y referencias. Pueden extender a otras clases, heredando sus atributos y operaciones.
- **EAttribute**: representa los atributos. Sus valores son de tipo primitivo: entero, booleano, cadena de caracteres, etc.
- **EReference**: representa relaciones binarias entre dos instancias de clases. La primera clase contiene o hace referencia a una o varias instancias de la segunda clase.

[1] http://www.eclipse.org/modeling/
[2] http://www.eclipse.org/modeling/emf/

- **EOperation**: hace referencia a las operaciones que pueden modificar el estado interno de un objeto de una clase.
- **EPackage**: representa un paquete que agrupa elementos del metamodelo. Los paquetes contienen clases y subpaquetes.

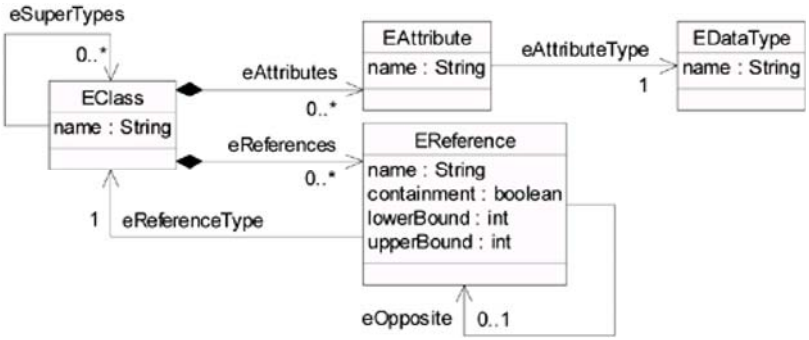

Figura 1-3. Núcleo de Ecore

1.4 Objetivos del trabajo

El objetivo principal de este trabajo es la integración de las herramientas MAST en el entorno RT-MDE.

En una primera aproximación se podría pensar en lanzar la interfaz gráfica de MAST desde el entorno RT-MDE, creando un agente de lanzamiento dedicado. En la Fig. 1-4 se puede apreciar cómo el modo de interacción actual con las herramientas MAST se basa en una interfaz gráfica de usuario que permite introducir los parámetros de entrada – siendo el parámetro principal el modelo MAST del sistema, conforme a la versión MAST 1.4 – y algunas opciones de configuración. Si el modelo consigue superar los numerosos chequeos de restricción internos, produce una salida con los resultados de planificabilidad de la aplicación que estamos analizando, en forma de modelo conforme a MAST Results 1.4.

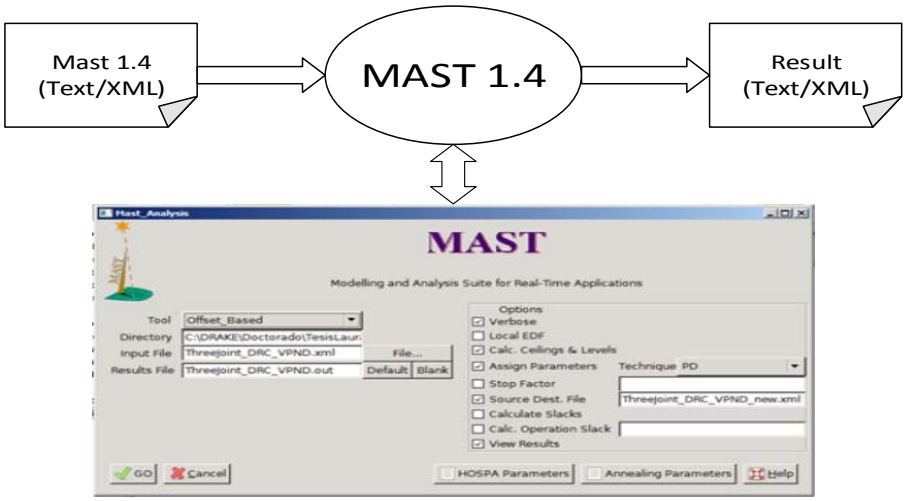

Figura 1-4. Ejecución de MAST 1.4

Sin embargo, esta opción no es adecuada por diferentes razones:

- Va en contra de la idea de entorno integrado, dónde toda la información (modelos, resultados, problemas, etc.) se muestra dentro de los recursos propios del entorno.

- Al ejecutarse MAST en consola del sistema operativo, no es posible hacer un seguimiento de la ejecución del proceso de análisis. Si lo que se pretende es que toda la información (advertencias, errores, resultados, seguimiento en consola) se gestione desde el entorno, no habría manera de llevar a cabo esta labor.
- Debería abordarse la tarea de realizar las conversiones de los modelos de entrada y resultados, ya que el entorno RT-MDE utiliza modelos Ecore conformes al metamodelo MAST 2.0, mientras que la herramienta MAST admite modelos XML o textuales conformes al metamodelo MAST 1.4.

Se requiere por tanto de una solución más elegante y no basada en el desarrollo de agentes de lanzamiento específicos para cada herramienta. En la Fig. 1-5 se muestra el modo de interacción con la herramienta MAST que se va a proporcionar. El entorno, como parte del proceso de desarrollo de una aplicación de tiempo real, deberá integrar las herramientas MAST, pero con capacidad de lanzamiento desde el propio entorno y mostrando los resultados a través de sus propios recursos (en este caso, los proporcionados por Eclipse).

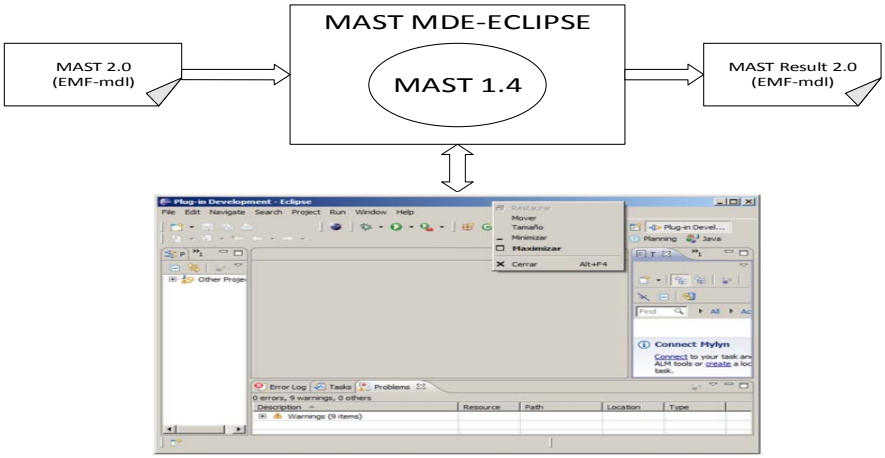

Figura 1-5. Lanzamiento de MAST desde RT-MDE

Una dificultad añadida para la integración de MAST proviene del hecho de que Eclipse está basado en Java, por lo que no puede interactuar de manera directa, sin una adaptación previa, con herramientas o aplicaciones escritas en otros lenguajes de programación. Este es el caso de MAST, que está desarrollado enteramente en Ada.

Además, el entorno RT-MDE se basa en utilizar como único formato de intercambio de modelos la serialización XMI de Ecore, por lo que será necesario disponer de mecanismos que permiten a las herramientas MAST trabajar con este formato de entrada y salida.

Con todo esto en mente, el objetivo final de la integración requiere abordar tres aspectos:

- Establecer el mecanismo de integración entre MAST y el entorno.
  Cómo se ha dicho anteriormente, el objetivo final de todo entorno integrado para diseño de sistemas de tiempo real es que toda la información manejada durante el proceso de desarrollo se muestre a través de los recursos propios de tal entorno. Para que esto sea posible es necesario dar soporte a los siguientes aspectos:
    - Lanzamiento de aplicaciones externas configuradas de acuerdo con la información contenida en una vista del entorno.
    - Mecanismos para que las aplicaciones puedan reaccionar a eventos generados por el operador sobre vistas de control desplegadas en el entorno.

- Utilización como terminal de salida para las aplicaciones de la vista de consola del entorno.
- Actualización desde la aplicación de la información visualizada en vistas de manifestación de estatus del entorno.

En lugar de abordarse estos aspectos desde el punto de vista específico de MAST, se ha optado por abordarlos de manera más genérica, esto es, definiendo una estrategia genérica para la integración de herramientas desarrolladas en lenguajes de programación diferentes de los soportados por el propio entorno, Ada en nuestro caso, o incluso herramientas Java que se ejecutan en distinta máquina virtual que el entorno.

- Procesar modelos MAST 2.0 – Ecore desde Ada.
  Para dar soporte a modelos de entrada y salida formulados de acuerdo a MAST 2.0 Ecore, será necesario desarrollar una librería o conjunto de módulos Ada que permitan procesar dichos modelos. En este caso también se ha optado por una solución más genérica, de manera que se va a desarrollar una librería Ada que permita leer y escribir información formalizada como modelo conforme a cualquier metamodelo Ecore y por tanto serializado según XMI de Ecore. La librería permitirá procesar modelos creando instancias suyas en memoria que puedan ser procesadas por las aplicaciones Ada correspondientes. Así mismo, deberá ofrecer capacidad para generar modelos de forma programática, de manera que puedan ser después fácilmente serializados a XMI. Como muestra la Fig. 1-6, dichas librerías se utilizarán posteriormente en la elaboración de un *parser* de modelos MAST 2.0 y un serializador de modelos de resultados MAST 2.0.

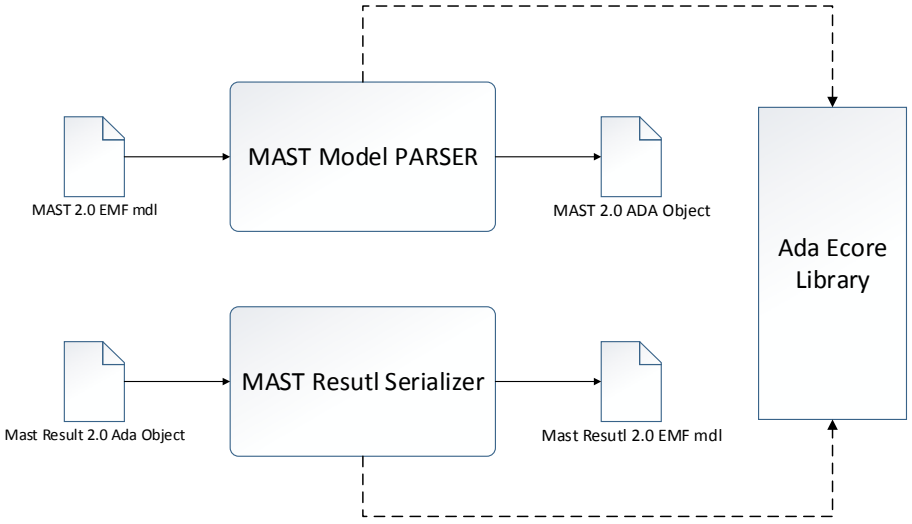

Figura 1-6. Módulos Ada de lectura y escritura de modelos MAST / Ecore

- Validación del mecanismo de integración para el caso de MAST.
  El mecanismo de integración definido deberá ser aplicado para la integración efectiva de MAST en el entorno. Para ello, además de proporcionar soporte para el procesamiento de modelos, será necesario analizar las posibles adaptaciones que deban ser realizadas en las herramientas MAST actuales.

1.5 Organización del trabajo

El resto del trabajo explica cómo se han abordado los objetivos expuestos:

- En la sección 2, se define la tecnología desarrollada que es necesaria para que el entorno RT-MDE pueda integrar herramientas desarrolladas en un lenguaje distinto de Java, en concreto para Ada. En primer lugar, se describe una librería que permitirá leer y escribir información en forma de modelos mediante los cuales, el entorno intercambia información con las herramientas externas. Posteriormente se define un mecanismo de comunicación entre el entorno y las herramientas para poder llevar a cabo el lanzamiento y seguimiento de las herramientas externas.
- En la sección 3, se implementa el mecanismo de la sección anterior para el caso concreto de MAST 1.4, para que puede integrarse en el entorno y se detallan los modelos de soporte utilizados en el mecanismo de comunicación.
- Finalmente, la última sección plantea las principales conclusiones del trabajo y las líneas de trabajo futuro.

2 Tecnología Ada para el entorno RT-MDE

2.1 Modelos Ecore para aplicaciones Ada

Con el objetivo de poder integrar herramientas codificadas en lenguaje Ada en el entorno RT-MDE, o en cualquier otro entorno donde se usen modelos Ecore, se ha desarrollado una librería Ada denominada *EM4Ada (Ecore Models for Ada)*, cuyas características se detallan a continuación. Se trata de una serie de módulos orientados a facilitar la lectura y escritura de modelos conformes a un metamodelo Ecore dado.

Las características del diseño de los módulos son:

- Son independientes de cualquier metamodelo, es decir, sin modificar su código pueden ser utilizados para leer o crear modelos, sea cual sea el metamodelo al que son conformes.
- La API proporcionada ha sido específicamente diseñada para ser utilizada por los programadores de aplicaciones Ada en base al conocimiento que tienen del metamodelo correspondiente.
- Los modelos son manejados en forma de un único fichero de texto, auto-contenido y formulado de acuerdo a la implementación que ofrece EMF del estándar XMI.
- Son módulos ligeros basados en almacenamiento temporal de la información y que liberan toda la memoria una vez que se han dejado de utilizar dentro de la aplicación.

2.1.1 Paquete EM4Ada

Es un paquete Ada 2012 que ofrece una interfaz para que desde una aplicación Ada se pueda leer la información incluida en un modelo conforme a un metamodelo *Ecore* y por tanto serializado a XMI. También ofrece capacidad para construir modelos de manera programática, de manera que puedan ser posteriormente serializados a XMI.

La interfaz ha sido específicamente diseñada para el caso en que el programador de la aplicación conozca el metamodelo al que es conforme el modelo que se procesa o genera. El programador conoce los tipos de objetos que existen en el modelo, y para cada uno de ellos conoce los identificadores y los tipos de sus atributos y de sus referencias.

A continuación, se describen las principales clases (tipos etiquetados en Ada) del paquete, que se muestran en la Fig. 2-1.

Clase *M_Object*

Describe un objeto del modelo, que representa una instancia de una de las clases definidas en el metamodelo. Una instancia de *M_Object* se identifica unívocamente por su atributo Id, que se utiliza para referenciar objetos dentro de un mismo modelo. Cada instancia de *M_Object* puede contener un conjunto de atributos, cada uno de ellos identificados por un nombre, y un conjunto de referencias también identificadas por su nombre.

M_Object presenta dos tipos principales de métodos:

- Métodos de lectura, que recuperan la información del objeto: *Get_Id(), Get_Class(), Get_Prefix_NS(), Get_Attribute(), Get_Multiple_Attribute(), Get_Reference(), Get_Multiple_Reference(), Get_All_References().*

Figura 2-1. Principales clases del paquete EM4Ada

- Métodos de escritura, que modifican la información del objeto: *Set_Attribute(), Add_Attribute(), Set_Multiple_Attribute(), Set_Reference(), Set_Multiple_Reference()*.

Clase *Resource*

Representa el recurso físico – fichero XMI – en el que se almacena un modelo de forma persistente. Permite acceder a los objetos contenedores raíces del modelo, a partir de los cuales se puede acceder a todo el resto de objetos.

Resource ofrece los siguientes métodos:

- *Load()*: Carga el fichero XMI en memoria. Cargar el fichero significa crear el árbol de instancias *M_Object* correspondientes al fichero XMI. En caso de que el fichero no exista, lanza la excepción *File_Not_Found_Exception* y si mientras se procesa el fichero ocurriese cualquier error, se lanza la excepción *Parser_Error_Exception*.
- *Get_Root_Objects()*: Devuelve un vector con todos los objetos raíces del modelo.
- *Add_Root_Object()*: Añade un objeto raíz al modelo. Este método se usará para crear modelos de forma programática.
- *Save():* Serializa el modelo almacenado en memoria (el modelo engloba todos los objetos raíces con sus correspondientes objetos anidados) en un fichero XMI. Si en la ruta indicada no existe el fichero, se crea.
- *Send():* Envía el documento XMI vía socket TCP.
- *Free():* Libera la memoria utilizada por el objeto y lo destruye.

Clase *M_Data_Value*

Describe cualquier valor escalar que puede ser asignado a un atributo de cualquier objeto. El valor es siempre representado internamente mediante una cadena de texto, pero se accede a él mediante funciones especializadas que retornan el correspondiente tipo primitivo. Todos los métodos de lectura elevan la excepción *Not_Valid_Data_Exception* si el valor almacenado no puede ser transformado al tipo primitivo requerido. También existen métodos de escritura que modifican el valor asociado a un objeto *M_Data_Value*, véase la Fig. 2-1.

2.1.2 Lectura y generación de modelos EMF/Ecore con EM4Ada

El uso de EM4Ada es relativamente sencillo, puesto que las clases involucradas (*Resource* y *M_Object*) ofrecen una interfaz bastante intuitiva y fácil de usar.

Con propósito ilustrativo se empleará un metamodelo muy simple, llamado *Family* y mostrado en la Fig. 2-2.

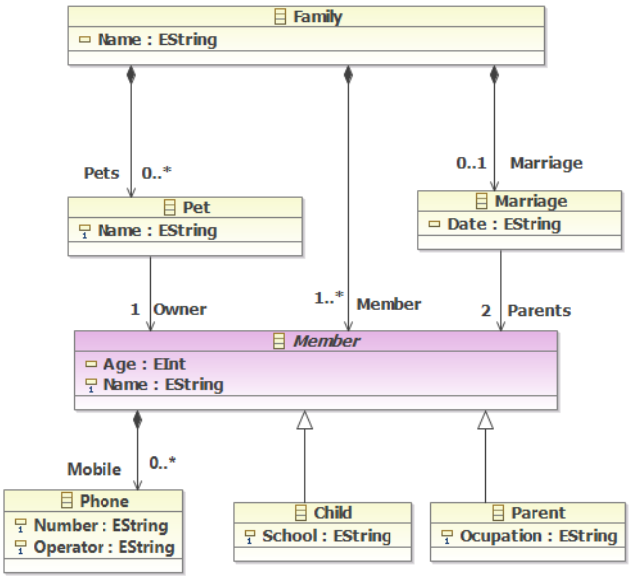

Figura 2-2. Metamodelo Family

El esquema de utilización típica de la librería EM4Ada en modo lectura es el que se muestra en la Fig. 2.3.

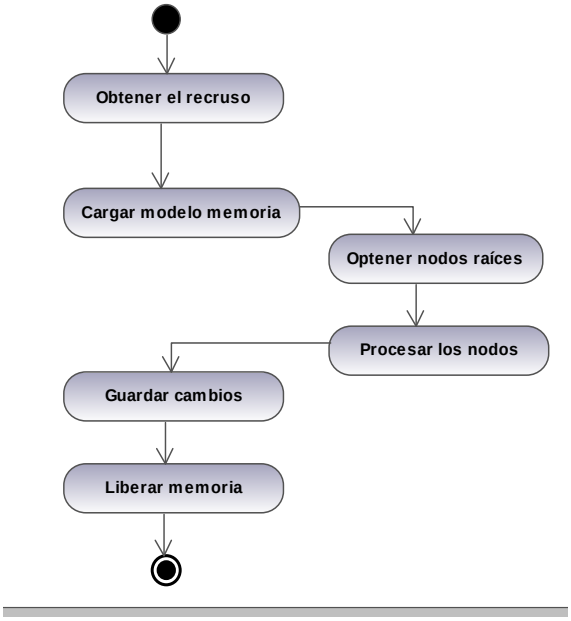

Figura 2-3. Proceso de lectura de un modelo Ecore-XMI

El código de la Tabla 2.1 es un ejemplo de procesamiento de un modelo conforme a *Family*. En primer lugar se crea una instancia de la clase *Resource* y se asocia a ella el fichero XMI. Su elemento raíz se asocia a una instancia de *M_Object (My_Family)*. A partir de este momento, los elementos del modelo pueden ser accedidos desde la raíz a través de los métodos de acceso a referencias y atributos, cuyos identificadores son conocidos por el programador que conoce el metamodelo. Asimismo, se puede modificar el propio modelo, como se puede ver en la Tabla 2.1 cuando se añaden nuevas referencias de tipo *Member* y *Pet* al objeto de tipo *Family* original. Tras la manipulación del modelo, se pueden guardar los cambios en un nuevo fichero, imprimir, enviar por la red, etc.

Tabla 2.1. Lectura y manipulación de un modelo Ecore-XMI con EM4Ada

```
procedure Test_Resource is
      Res : constant Resource_Ref:= Create_Resource;
      My_Family:M_Object_Ref;
      My_Pet:M_Object_Ref;
      Member_X:M_Object_Ref;
begin
      -- cargar el modelo en memoria
      Res.Load(To_Unbounded_String("Family.xmi"));
      -- obtener el elemento raiz
      My_Family:= Res.Get_Root_Objects.Element(0);
      -- mostrar el nombre de la familia
      Ada.Text_IO.Put_line(My_Family.Get_Attribute("Name"));

      -- añadir un nuevo miembro a la familia
      Member_X:= Create_Object(Class =>"Member");
      Member_X.Set_Attribute("Name","Daniel");
      Member_X.Set_Attribute("Age","24");
      My_Family.Add_Reference("Members",Member_X, true);
```

```
      -- añadir una nueva mascota a la familia
      My_Pet:= Create_Object(Class =>"Pet")
      My_Pet.Set_Attribute("Name","Bracco");
      My_Pet.Set_Reference("Owner",Member_X, false);
      My_Family.Add_Reference("Pets",My_Pet,Is_Contaiment=> true

      -- enviar por la red
      Res.send("192.168.1.104",39002);
      -- serializar a XMI
      Res.Save("Modified_Family.xmi");
      Free(Res);

end Test_Resource;
```

El esquema de utilización típica de *EM4Ada* en modo escritura (generación) es el que se muestra en la Fig. 2.4.

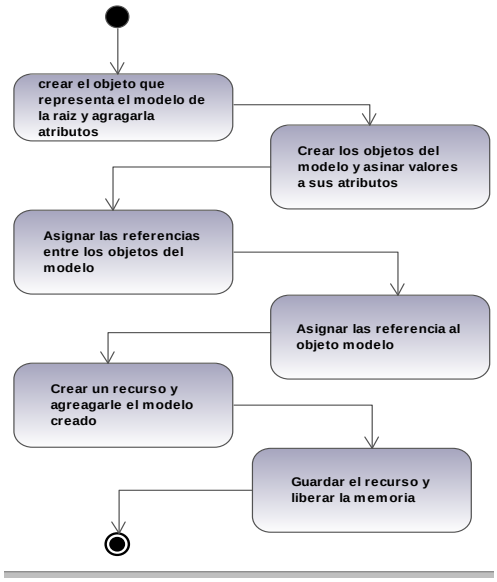

Figura 2-4. Generación de un modelo y posterior serialización con EM4Ada

El código de la Tabla 2.2 es un ejemplo de generación y posterior serialización de un modelo conforme al metamodelo *Family*.

Tabla 2.2. Generación de un modelo y posterior serialización a XMI con EM4Ada

```
procedure Generate_Model is
     family_x: M_Object_Ref;
     member_a, member_b: M_Object_Ref;
     pet_z: M_Object_Ref;
     Res: Resource_Ref;
begin
     -- crear una familia, miembros y mascotas
     family_x:= Create_Object(Class =>"Family");
     member_a:= Create_Object(Class =>"Parent");
     member_b:= Create_Object(Class =>"Child");
     pet_z   := Create_Object(Class =>"Pet");
     -- insertar un nombre para la familia
     family_x.Set_Attribute("Name","Las Cristinas");

     -- insertar atributos para los miembros
     member_a.Set_Attribute("Name","Ana Cristina");
```

```
      member_a.Set_Attribute("Age","38");
      member_a.Set_Attribute("Ocupation","Enfermera");
      member_b.Set_Attribute("Name","Rosa Cristina");
      member_b.Set_Attribute("Age","11");
      member_b.Set_Attribute("School","IES Las Llamas");

      -- insertar nombre y dueño para la mascota
      pet_z.Set_Attribute("Name","Cristian");
      pet_z.Set_Reference("Owner", member_a, false);

      -- agregar miembros y mascotas a la familia
      family_x.Add_Reference("Member", member_a, true);
      family_x.Add_Reference("Member", member_b, true);
      family_x.Add_Reference("Pets", Pet_z, true);

      -- crear el recurso y serializar
      Res := Create_Resource;
      Res.Add_Root_Object(family_x, "family", "Family");
      Res.Save("familia_las_cristinas.txt");
      Free(Res);

end Generate_Model;
```

Una propuesta de uso de la librería EM4Ada

La clase *Mapped_Class* mostrada en la Tabla 2.3 es muy importante para facilitar la lectura de modelos desde la fuente (fichero) y poder *parsear* el modelo sin demasiada complejidad. Es aconsejable su uso en caso de que no dispongamos de una estructura predeterminada ya creada en Ada.

Por cada clase del metamodelo a procesar podemos definir una clase que extienda a *Mapped_Class* y que incorpore métodos de acceso y escritura de todos sus atributos y referencias. Dado que la clase *Resource* ya nos proporciona toda la colección de instancias *M_Object* leídas, lo único que hay que hacer para dar soporte a cada instancia del modelo es comprobar el atributo *Class* de cada instancia *M_Object* y crear una instancia de la clase mapeada correspondiente. Los valores no hace falta copiarlos uno a uno, basta con hacer uso del método *Set_Object*, el cual asocia una instancia de *M_Object* a la instancia de la clase.

El código de la Tabla 2.4 muestra una clase especializada de *Mapped_Class* que simplemente incorpora métodos que operan en la instancia de *M_Object* asociada. Como ejemplo, la Tabla 2.5 muestra la implementación correspondiente al método que retorna el valor del atributo *Age()*.

Tabla 2.3. Clase genérica que representa cualquier objeto del modelo

```
type Mapped_Class is abstract tagged private
     function Object(Self:in Mapped_Class)return M_Object_Ref;
     procedure Set_Object(Self:inout Mapped_Class;
                          Object: M_Object_Ref);
private
     type Mapped_Class is tagged record
       Object :M_Object_Ref;
     end record;
```

Tabla 2.4 Clase Ada que mapea la clase Member del modelo Ecore usando EM4Ada

```
type Member is new Mapped_Class with null record;
```

```
function Age(Self : in Member) return String;
procedure Set_Age(Self: in out Member, Age : Natural);

function Name(Self : in Member) return String;
procedure Set_Name(Self: in out Member, Name : String);
```

Tabla 2.5. Implementación del método *Age()*, que devuelve el atributo *Age*

```
function Age(Self :in Member)return Natural is
begin
     if Self.Object.Get_Attribute("Age")= null then
         return Natural'First;
         -- return default value or raise exception
         -- if a value is required
     else
         return Self.Object.Get_Attribute("Age").Get_Integer;
     end if;
     exception when others=> return Natural'First;
end Age;
```

2.2 Estrategia de implementación

2.2.1 Análisis de la formulación ECORE-XMI

La especificación XMI (*XML Metadata Interchange*) de OMG [8] es un estándar que define cómo serializar modelos y meta-modelos en XML. El intercambio basado en XMI de artefactos de modelado es esencial en entornos distribuidos y heterogéneos. EMF ofrece una implementación de este estándar, definiendo cómo tanto un metamodelo Ecore como sus modelos instancia son mapeados a formato XML acorde a la especificación XMI. Para poder desarrollar el módulo EM4Ada fue necesario analizar el formato de tal serialización de Ecore a XMI, siendo de especial importancia la manera en que son mapeados los elementos más complejos, como las referencias cruzadas entre objetos. Para mostrar los principales resultados de dicho análisis se utiliza como ejemplo el documento XMI correspondiente a una instancia del metamodelo *Family* mostrado en la sección anterior, y que se muestra en la Tabla 2.6.

Análisis de atributos y referencias

Los atributos – *EAttribute* en *Ecore* - se mapean como en XML convencional, por ejemplo, en el caso del ejemplo el objeto *family* tiene un atributo *Name* cuyo valor es “Familia Vidal Nieto”. Las referencias, - *EReference* en *Ecore* - pueden ser de dos tipos, asociación y composición:

- Las referencias por composición se formulan como elementos anidados en el objeto contenedor, como ocurre con las referencias *Member*, *Pets* y *Marriage* en *Family*.
- Por el contrario, las referencias por asociación se formulan como atributos simples, donde el valor asignado es un *string* que referencia a los objetos correspondientes siguiendo siempre el siguiente formato: *//@Reference*. Como ejemplo, en la Tabla 2.6, el único objeto *Marriage* tiene una referencia por asociación denominada *Parents* que referencia a dos objetos de tipo *Member* del modelo (//@Member.0 y // @Member.1).

Tabla 2.6. Ejemplo de modelo conforme al metamodelo *Family* en formato Ecore-XMI

```
<?xmlversion="1.0"encoding="ASCII"?>
<family:Family xmi:version="2.0"
   xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:family="http://family/1.0"
   xsi:schemaLocation="http://family/1.0 ../Mmodel/Family.ecore"
   Name="Familia Vidal Nieto">

   <Member xsi:type="family:Parent" Name="Lorenzo Vidal Elche"
           Age="50" Ocupation="Enfermero">
      <Mobile Number="000 000 000" Operator="Vodafone"/>
      <Mobile Number="222 444 000" Operator="Yoigo"/>
   </Member>
   <Member xsi:type="family:Parent" Name="Natividad  Nieto Lozano"
           Age="36" Ocupation="Contable"/>
   <Member xsi:type="family:Child" Name="Silverio Vidal Nieto"
           Age="4" School="IES Santa Clotilde"/>
   <Pets Name="Bracco" Owner="//@Member.1"/>
   <Marriage Parents="//@Member.0 //@Member.1" Date="10-02-2006"/>

</family:Family>
```

Determinación de la clase de un objeto

Utilizando la información formulada de manera explícita en el documento XMI no es posible conocer la clase de todos los objetos del modelo, lo cual es un inconveniente cuando se trabaja con modelos sin contar con la información del metamodelo.

Por ejemplo, se sabe por el meta-modelo de la Fig. 2-3 que la referencia *Pets* del objeto *Family* pertenece a la clase *Pet*. En cambio, tan sólo consultando la formulación XMI de la Tabla 2.6 no sería posible conocer la clase a la que pertenece.

Las reglas que se siguen para poder determinar la clase de cada instancia incluida en un modelo XMI son las siguientes:

- Las instancias de clases concretas, que además no sean subclases, se representan como elementos de XMI incluyendo únicamente sus atributos y referencias de la manera anteriormente descrita. En nuestro ejemplo, los objetos de la clase *Pet* seguirían este patrón.
- Las clases abstractas no se pueden instanciar, por ejemplo, no pueden existir en el modelo instancias de la clase *Member*.
- Aquellos objetos que corresponden a subclases en un árbol de herencia necesitan un atributo adicional *xsi:type* que sirve únicamente para determinar la subclase del objeto cuando se serializa a XMI. En nuestro ejemplo, los objetos de tipo *Member* necesitan un valor para el atributo *xsi:type* para especificar la subclase: *Parent* o *Child*.
  A veces se utiliza el atributo *xmi:type* en lugar de *xsi:type*. No hay diferencias significativas, pero *xmi:type* admite modelos con herencia múltiple.

Análisis de identificadores

En la formulación XMI de un modelo Ecore, los elementos no llevan identificadores de manera explícita. Cada elemento se identifica de forma unívoca según su etiqueta (en este caso el

nombre de la referencia a la que pertenece) y el orden que ocupa en la lista de los de su misma etiqueta en el elemento contenedor al que pertenece. Esto se conoce como fragmento URI. En nuestro ejemplo, el objeto de tipo *Family* (el objeto raíz) tiene tres objetos asignados a su referencia *Member*, cuyos identificadores relativos al objeto contenedor son respectivamente: *@Member.0*, *@Member.1* y *@Member.2*. El objeto *@Member.0* tiene a su vez anidados dos objetos de tipo *Mobile*, cuyos identificadores son *//@Member.0/@Mobile.0* y *//@Member.0/@Mobile.1*, respectivamente.

La ausencia de un atributo explícito "id" que actúe como campo de identificación para los objetos es debida al modo de uso de la interfaz *XMIResource*, utilizada para serializar modelos EMF/Ecore a XMI. El formato de serialización se basa en la especificación XMI 2.0 de la OMG. Se puede especificar la codificación XML que se utilizará al guardar el recurso mediante el método *setEncoding*.

XMIResource no crea identificadores de forma automática, se puede especificar de forma manual como identificador un atributo definido. Sin embargo, se recomienda que se utilice fragmentos de URI debido a que su uso reduce el tamaño de los ficheros XMI y el consumo de memoria. Este es el tipo de codificación que usa la plataforma Eclipse por defecto cuando guarda un modelo *Ecore* en formato XMI.

La Fig. 2-5 muestra un modelo con identificadores basados en URI, es decir, la posición relativa respecto al nodo raíz, cuando se ordenan los elementos de la misma etiqueta. En el modelo de la Fig. 2-6 los identificadores son únicos y explícitos en un atributo "id" del elemento.

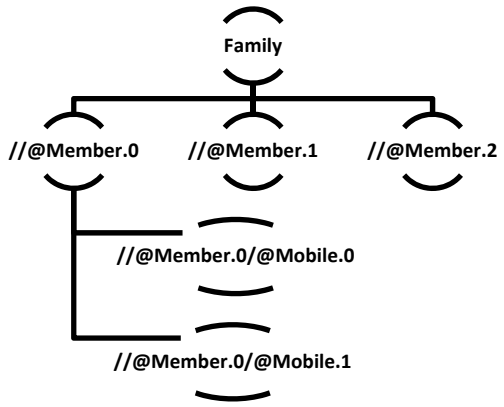

Figura 2-5 Modelo con identificadores basados en URI

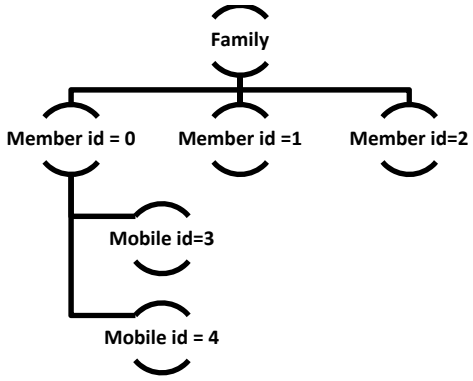

Figura 2-6. Modelo con identificadores explícitos predefinidos

2.2.2 Soporte Ada para SAX

SAX (*Simple API for XML*)[1] es un API basado en eventos para leer documentos XML e implementado por diversos *parsers*, entre ellos el de Oracle para Java. De hecho, SAX fue originalmente definido como un API Java y está principalmente orientado a analizadores escritos en este lenguaje. En cualquier caso, SAX se ha portado a la mayoría de lenguajes orientados a objetos, como C++, Python, Ada, etc.

SAX es bastante atípico entre los APIs de XML, pues consiste en un modelo basado en eventos en lugar de basado en árbol, como por ejemplo el API DOM (*Document Object Model*)[2]. Según un documento XML va siendo leído, envía al programa información del documento en tiempo real y así el documento se presenta cada vez al programa de una pieza, de principio a fin. Es posible guardar los fragmentos de interés hasta que el documento entero haya sido leído o procesar la información tan pronto como es recibida, no siendo necesario esperar a que se lea el documento completo antes de procesar los datos en su parte inicial y, lo más importante de todo, no es necesario que el documento completo resida en memoria. Esto hace de SAX el API apropiado para documentos muy extensos que pueden desbordar la memoria disponible.

El soporte que Ada ofrece en lo relativo a SAX viene representado por la clase abstracta *SAX.Readers*. Ésta representa un analizador SAX que genera distintos eventos. En cada uno de ellos se hace una llamada al método manejador correspondiente. Se utiliza el mismo nombre tanto para los eventos como para los manejadores:

- **Start_Document**. Es el manejador del evento de inicio de documento. Se llama una vez y por lo general, se utiliza para inicializar las estructuras de datos. No tiene parámetros.
- **End_Document.** Es el manejador del evento de fin de documento y también se llama una vez. No tiene parámetros y se utiliza típicamente para liberar memoria.
- **Start_Element.** Es el manejador del evento inicio de un elemento en el documento XML. Tiene como parámetros el nombre del elemento, información acerca del espacio de nombres y la lista de atributos del elemento.
- **End_Element.** Es el manejador del evento fin de un elemento. Tiene como parámetros el nombre del elemento y información acerca del espacio de nombres.
- **Characters**. Maneja el evento que se produce cuando se encuentra texto fuera de la declaración del elemento. El texto se recibe como argumento.

Estos manejadores no están implementados en la clase abstracta *SAX.Readers*, por tanto, para usar SAX en Ada, se debe extender la clase *SAX.Readers* e implementar estos métodos con la lógica apropiada a la aplicación que se esté desarrollando.

2.2.3 Implementación Ecore_Sax_Reader

La Fig. 2-7 muestra la clase *Ecore_Sax_Reader*, extensión de *SAX.Readers* que se ha diseñado para la lectura de modelos Ecore. Sus métodos principales son:

- *Initialize():* Inicializa el objeto *Ecore_Sax_Reader*.

[1] www.saxproject.org
[2] http://www.w3.org/DOM/

- *Parse()*: Recibe como entrada un fichero XMI. Su función es la lectura de dicho fichero, capturando los distintos eventos que se han descrito en la sección anterior. La captura de un evento implica la llamada automática del correspondiente método manejador.
- Todo el conjunto de métodos manejadores indicados anteriormente.

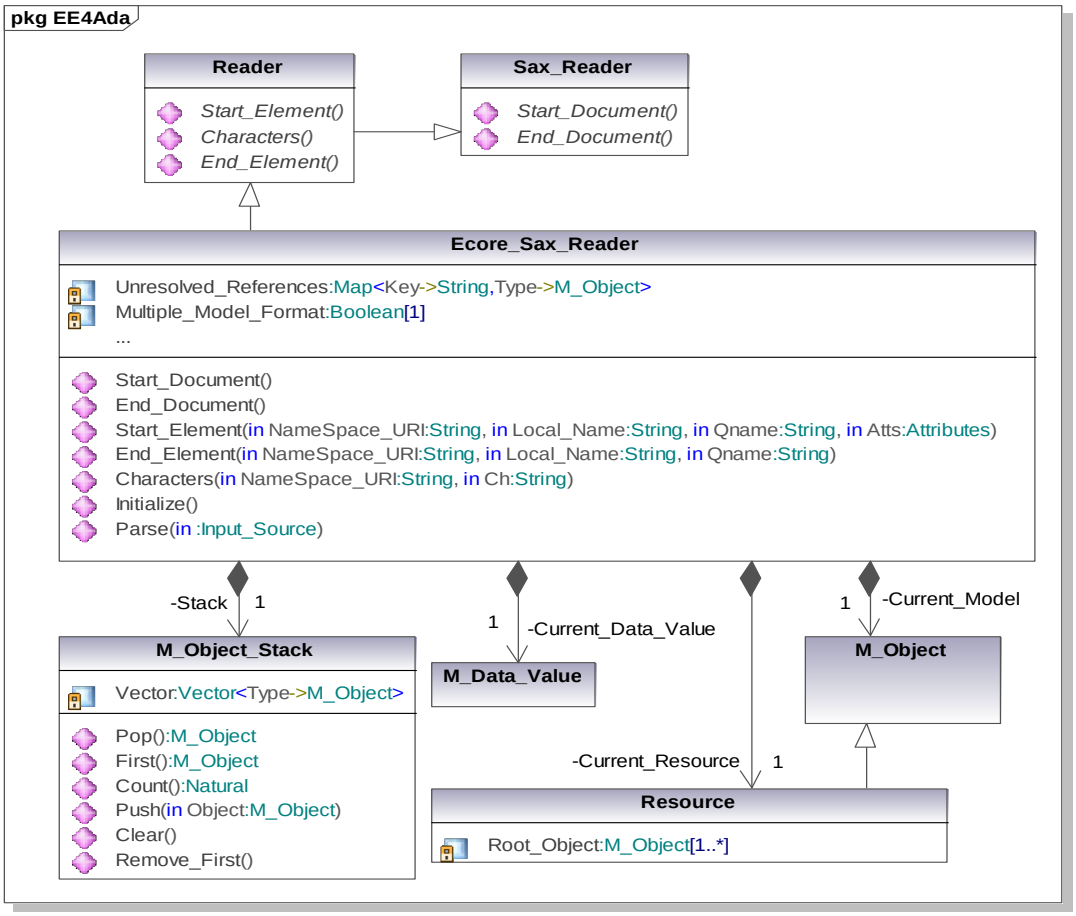

Figura 2-7. API de *Ecore_Sax_Reader*

Procesamiento. Lectura y resolución de referencias

SAX, en contraste con DOM, no carga de inicio todo el árbol de objetos en memoria. Eso dificulta la tarea de manipulación de los elementos del fichero XML que se está procesando, aunque a cambio, el procesamiento es más rápido, además del bajo consumo de memoria.

Dado que los objetos pueden referenciar a otros, a la hora de leer el fichero puede darse el caso de que un objeto haga referencia a otro que todavía no ha sido leído. Esta referencia no se puede asignar puesto que no tenemos todos los datos en memoria en el momento adecuado. Habitualmente se recorre dos veces el fichero, primero se cargan todos los objetos en memoria y en el segundo recorrido se resuelven las referencias.

Aquí se va a proponer un método que evita esta segunda vuelta, de modo que se pueda leer y asignar referencias en un único recorrido del documento. El objetivo es que, cuando se termina de procesar un determinado objeto, éste tenga todas sus referencias asignadas.

Para ello se usa un mapa auxiliar – *Unresolved_References* -, de objetos *M_Object* utilizando como índice el atributo ID del objeto (este ID corresponde con la URI del elemento en el espacio de nombres). Se van añadiendo a dicho mapa los objetos referenciados que no han sido expandidos. Cuando se está procesando un elemento nuevo, se comprueba si ha sido

referenciado con anterioridad, es decir, si existe un objeto en el mapa *Unresolved_References* con la misma URI. En caso afirmativo, la información del elemento XMI se guarda en el objeto de la lista, en caso contrario se debe crear un nuevo objeto donde guardar su información. En las composiciones no se presenta ningún problema, pues al ser un algoritmo recursivo, siempre se termina por expandir primero los nodos más profundos de cada rama.

En la situación de la Fig. 2-8, el nodo N3, tiene tres referencias, N2, N4 y N7:

- Para la referencia de N3 a N2, como N2 ya estará en el árbol, la asignación se hace recuperando el objeto directamente del árbol.
- Para la referencia de N3 a N4, como se trata de una composición, se creará un nodo hijo y la asignación es trivial.
- Para la referencia de N3a N7, primero se buscará N7 en el árbol, y como no está, se buscará en *Unresolved_References*. Como tampoco está en este mapa (N7 no ha sido referenciado con anterioridad), entonces se crea un nuevo nodo (*M_Object*) que se añade a *Unresolved_References* y posteriormente es asignado como referencia a N3. El nuevo nodo creado ya está apuntado por N3, pero le faltan los demás atributos que se van a completar cuando se llegue al elemento correspondiente del fichero, es decir, el elemento cuya URI coincida con la URI de N7.
- Al procesar el nodo N7, como ya está en el mapa *Unresolved_References*, se recupera de éste y se le completa con los atributos del elemento. Esto garantiza que se están completando los atributos del objeto referenciado por N3 y N5. Seguidamente deberá eliminarse el objeto de *Unresolved_References*, pues ya se ha resuelto la referencia. Si algún nodo posterior lo volviera a referenciar, no habrá problemas en encontrarlo, puesto que ya estará en el árbol.
- Si al final del documento hay algún elemento en *Unresolved_References*, entonces ha ocurrido un error, pues algún elemento ha sido referenciado pero no existe ningún elemento en el documento que coincida con su URI.

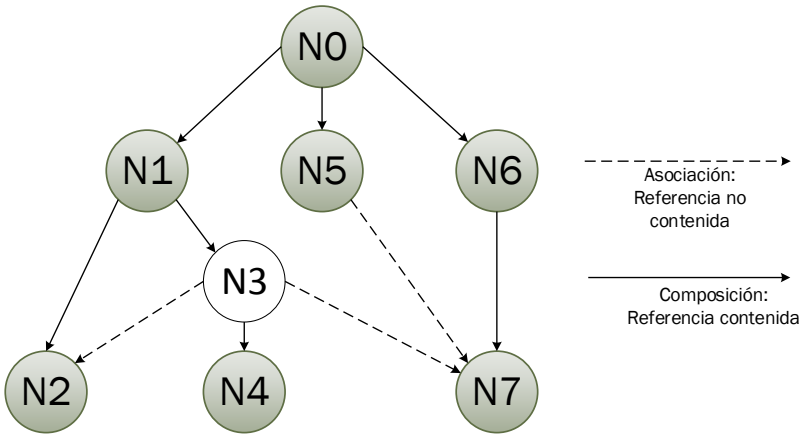

Figura 2-8 Tipos de referencia y su resolución

2.2.4 Futuras extensiones (lectura del metamodelo)

Al no contarse con el metamodelo almacenado en memoria, existen algunos inconvenientes que deben ser tenidos en cuenta a la hora de procesar modelos.

En el caso del software de procesamiento de un modelo Ecore existen algunos problemas a la hora de recuperar la información de un elemento cuando existe un valor por defecto de un atributo opcional.

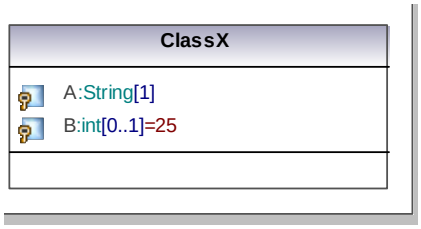

Figura 2-9. Metamodelo Simple

Supongamos el metamodelo de la Fig. 2-9, los objetos:

```
Ob1: ClassX =("MyString",25)
Ob2: ClassX =("MyString")
```

son equivalentes según la forma en que ECORE guarda la información de los modelos, que sería:

```
<ClassX A="MyString"/>
```

Ahora bien, a no ser que considerar tener el valor 25 en el atributo (B) sea equivalente (lo mismo para todas las clases con este tipo de problema) a que no existe, estamos ante un problema ambiguo que el software de procesamiento, sin la información contenida en el metamodelo, no sabría cómo procesar. El criterio utilizado en este proyecto es ignorar los valores no incluidos en el modelo. No obstante, en la clase que mapea esta clase en el programa Ada, deberá devolverse el valor por defecto en caso de que dicho atributo contenga un valor nulo.

La ambigüedad consiste en que la clase *ClassX* tiene un atributo obligatorio *A*, y otro opcional B con un valor por defecto, por lo que habrá instancias que tengan definido el atributo B y otras que no. La forma en que Ecore serializa los modelos no diferencia entre las instancias que definen el atributo opcional *(B)*, con el valor por defecto, y las que no lo definen. Esta información es consultada en el metamodelo en caso de que la variable exista, pero no se serializa en XMI.

Así mismo, el hecho de no contar con el metamodelo cargado en memoria, facilita en gran medida la posibilidad de introducir inconsistencias en los modelos introducidos de manera programática, ya que toda la responsabilidad de conocer los atributos y referencias de un determinado objeto recae en el programador.

Las situaciones descritas pueden comprometer la consistencia de la información a la hora de procesar/generar un modelo EMF/Ecore. En este proyecto no se ha abordado este problema y se deja como trabajo futuro que el software de procesamiento cargue previamente el metamodelo antes de procesar el modelo, así estará disponible toda la información necesaria para la toma de decisión de ciertas situaciones ambiguas.

2.3 Infraestructura de soporte para la interacción entre aplicaciones externas y los recursos del entorno

Tal y como se explicó en la introducción, el segundo aspecto al que hay que dar soporte para poder desarrollar el entorno RT-MDE orientado a MAST e implementado sobre Eclipse es la integración de MAST con el entorno, desarrollando soluciones que permitan realizar el lanzamiento y seguimiento de la ejecución de MAST desde el propio entorno. En lugar de abordar el problema desde un punto de vista exclusivo de MAST, se plantea una infraestructura de soporte para la interacción entre el entorno y cualquier artefacto externo a éste, ya sea en máquina local o remota. Se considera artefacto externo a cualquier recurso software utilizado

desde el entorno que no se ejecute en el mismo espacio de memoria, es decir, en la misma máquina virtual Java.

La Fig. 2-10 muestra las vistas más importantes del entorno en lo que respecta a este proyecto.

a. *Console*. Marco de tipo visor donde se muestran los avisos textuales que el artefacto envía al entorno y donde así mismo se refleja la información que se envía al artefacto desde teclado.
b. *Problems*. Marco de tipo visor destinado a mostrar en forma tabular los problemas detectados durante la ejecución del artefacto. Para cada problema, se expresa a través de un conjunto de campos información como la naturaleza y severidad (error, advertencia o recomendación) del problema, la localización del elemento en el que se ha detectado tal problema y consejos de cómo puede ser subsanado.
c. *ToolOutline*. Marco cuya función es mostrar la constitución de una herramienta, dando acceso a la información que tiene asociada y permitiendo controlar su ejecución. Cuando una herramienta es invocada, se hace visible este marco y se muestra en él su información asociada, en particular la secuencia de tareas de que está compuesta. La vista permanece con la información de la herramienta mientras se está ejecutando y se vacía a su conclusión.

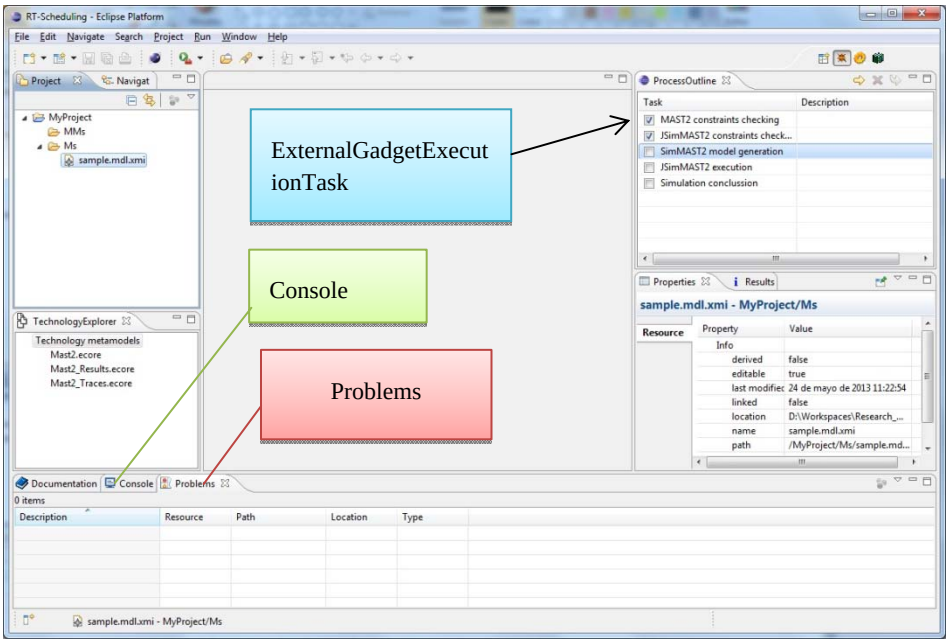

Figura 2-10. Vista del entorno RT-MDE

2.3.1 Comunicación entre artefactos externos y el entorno

Aplicando una estrategia estrictamente dirigida por modelos, la comunicación entre RT-MDE y los artefactos externos va a consistir únicamente en el intercambio de información formalizada como modelos XMI (codificados en ASCII) conformes a metamodelos Ecore conocidos por ambas partes. Teniendo esto en cuenta, existen varias alternativas para implementar la comunicación entre el entorno y los artefactos:

- Ficheros del disco: Ineficientes, y sólo válidos cuando el entorno y el artefacto están en la misma máquina, gobernados por un sistema operativo común.

- Pipes del sistema operativo: Son dependientes de la plataforma y válidos sólo para sistemas monoprocesadores.
- Sockets TCP: Más eficientes y válidos para cualquier situación. En concreto, permiten que el artefacto esté ubicado incluso en una máquina y sistema operativo diferentes de los del entorno. Es el mecanismo que se ha decidido utilizar en este proyecto.

La Fig. 2-11 resume la estrategia de interacción definida. El artefacto externo (*Gadget*) será accedido desde un *Wrapper* que se ejecuta como demonio en la máquina en la que se encuentra el artefacto. Un mismo *Wrapper* podrá gestionar varios artefactos disponibles en la misma máquina. Por su parte, el entorno guardará una lista de artefactos externos junto con cierta información sobre la forma de acceder a los mismos, a fin de poder ser utilizados durante el proceso de desarrollo. Para poder lanzar un artefacto, el entorno enviará al *Wrapper*, a través de un puerto predefinido, un mensaje (modelo) de descubrimiento con suficiente información para determinar el artefacto que se solicita al *Wrapper* que lo gestiona. Si el *Wrapper* localiza el *Gadget* solicitado y comprueba que está en disposición de ser ejecutado, le envía un mensaje (modelo) de conexión aceptada al entorno, el cual enviará el resto de información necesaria para la ejecución del *Gadget*.

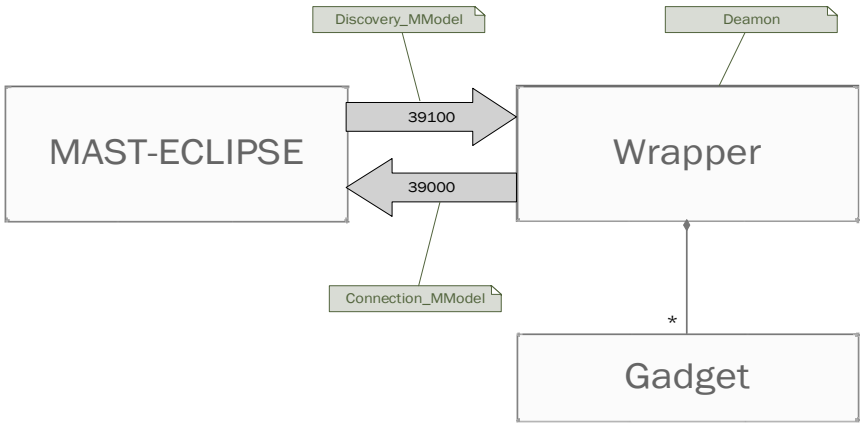

Figura 2-11. Elementos básicos de la comunicación entre el entorno y un artefacto externo

La Fig. 2-12 muestra la estructura definida para que un *Wrapper* pueda dar soporte a esta estrategia. El *Wrapper* consta de los siguientes elementos:

- *Discovery_Agent*: Es un elemento activo que se queda esperando solicitudes de lanzamiento de cualquier artefacto gestionado por el *Wrapper*.
- *Discovery_Model*: Cada solicitud de lanzamiento implica la recepción de un modelo conforme al metamodelo *Discovery_MModel* que incluye, entre otras cosas, información del artefacto que se está solicitando.
- *Connection_Model*: Tras analizar el modelo conforme a *Discovery_MModel*, el *Wrapper* deberá devolver un modelo de conexión conforme a *Connection_MModel* indicando el estado del artefacto solicitado.
- *Gadgets*: Lista de artefactos gestionados por el *Wrapper*.

El significado de los campos de los metamodelos *Discovery_MModel* y *Connection_MModel* se explica en detalle en la sección 2.3.2.

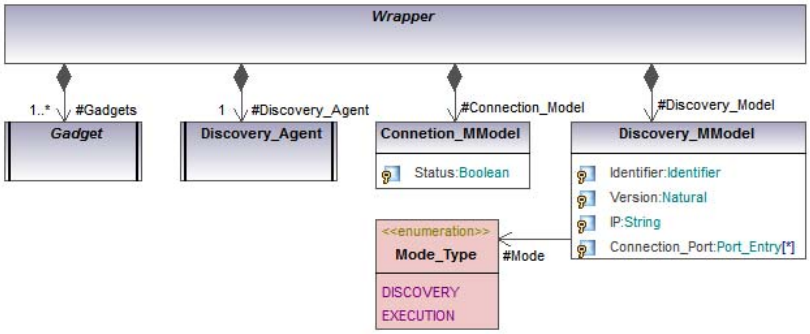

Figura 2-12. Estructura del *Wrapper* que gestiona artefactos externos al entorno

Cada *Gadget* corresponde con uno de los artefactos gestionados por el *Wrapper*. Su estructura se muestra en la Fig. 2-13. Al igual que ocurre con el *Wrapper*, la estructura de un *Gadget* consta por un lado de los de modelos de intercambio de información con el entorno (intercambiados a través del *Wrapper*), y por otro de los elementos activos que se encargan de la ejecución del artefacto y de la gestión de los modelos.

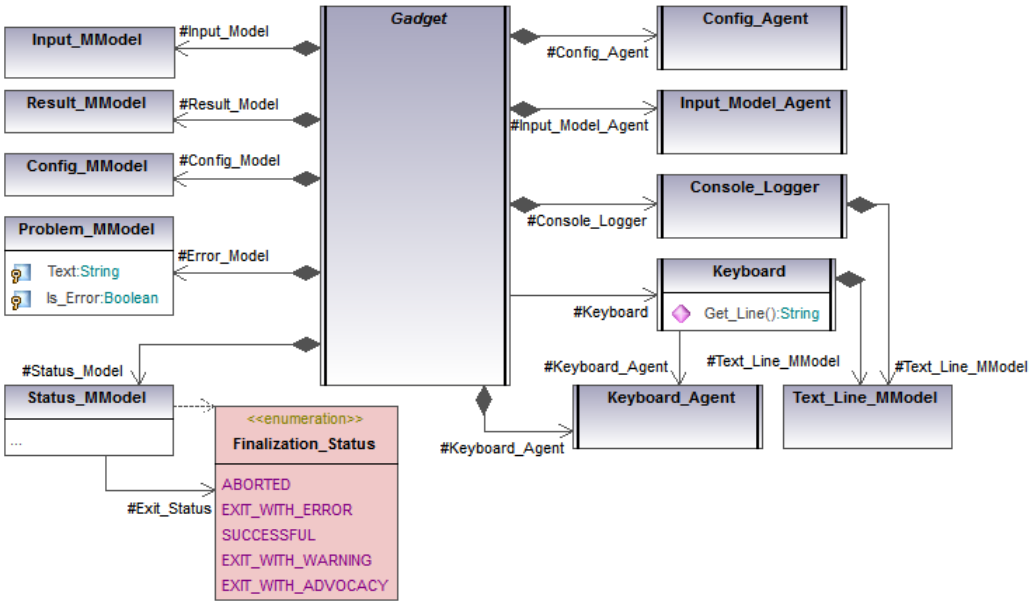

Figura 2-13. Estructura del *Gadget* externo al entorno

Los modelos que intercambia el *Gadget* con el entorno son:

- *Input_Model*: Modelo de entrada del artefacto externo, conforme al metamodelo *Input_MModel*. Contiene elementos propios del dominio considerado. Correspondería al modelo de entrada MAST en el caso de RT-MDE.
- *Config_Model*: Modelo de configuración, conforme al metamodelo *Config_MModel*. Contiene la información necesaria para la ejecución del artefacto según el modelo de entrada. Es, por tanto, dependiente del artefacto considerado.
- *Result_Model*: Modelo de resultados que el artefacto produce tras procesar el modelo de entrada, conforme al metamodelo *Result_MModel*. También contiene elementos del dominio considerado. Se trataría del modelo de resultados del análisis en el caso de MAST.
- *Status_Model*: Modelo con la información sobre el estado del artefacto al finalizar su ejecución, conforme al metamodelo *Status_MModel*.

- *Problem_Model*: Modelo de problemas, conforme a *Problem_MModel*. Contiene información sobre cualquier problema que se haya producido en el transcurso de la ejecución del artefacto.

En resumen, los metamodelos *Input_MModel*, *Result_MModel* y *Config_MModel* deberán ser definidos específicamente para cada nuevo artefacto, mientras que los metamodelos *Status_MModel* y *Problem_MModel* son comunes a todos ellos. El significado de sus campos se explica en detalle en la sección 2.3.2.

Los elementos activos del *Gadget* son:

- *Config_Agent:* Encargado de recibir el modelo de configuración.
- *Input_Model_Agent:* Encargado de recibir el modelo de entrada.
- *Console_Logger:* Encargado de enviar al entorno la salida que produce el artefacto.
- *Keyboard_Agent*: Encargado de recibir líneas de consola como entrada de teclado.
- *Keyboard*: Elemento que simula el funcionamiento de la entrada estándar. Las solicitudes de datos en la entrada estándar por parte del *Gadget* se deberán realizar a través de los métodos que ofrece *Keyboard*. Si *Keyboard_Agent* recibe una línea de entrada de teclado, el elemento activo *Keyboard* se encarga luego de entregarlo al *Gadget*, el cual deberá estar bloqueado a la espera de una línea de entrada en la entrada estándar según el contexto en el que esté ejecutando.

A modo de resumen, se puede observar en la figura 2-14 los modelos intercambiados entre el entorno y el artefacto externo. En el siguiente apartado se analizan en detalle los metamodelos que deben conocer ambas partes para que la información pueda ser interpretada correctamente. También se describen las distintas fases por las que pasa la ejecución del artefacto externo desde el entorno.

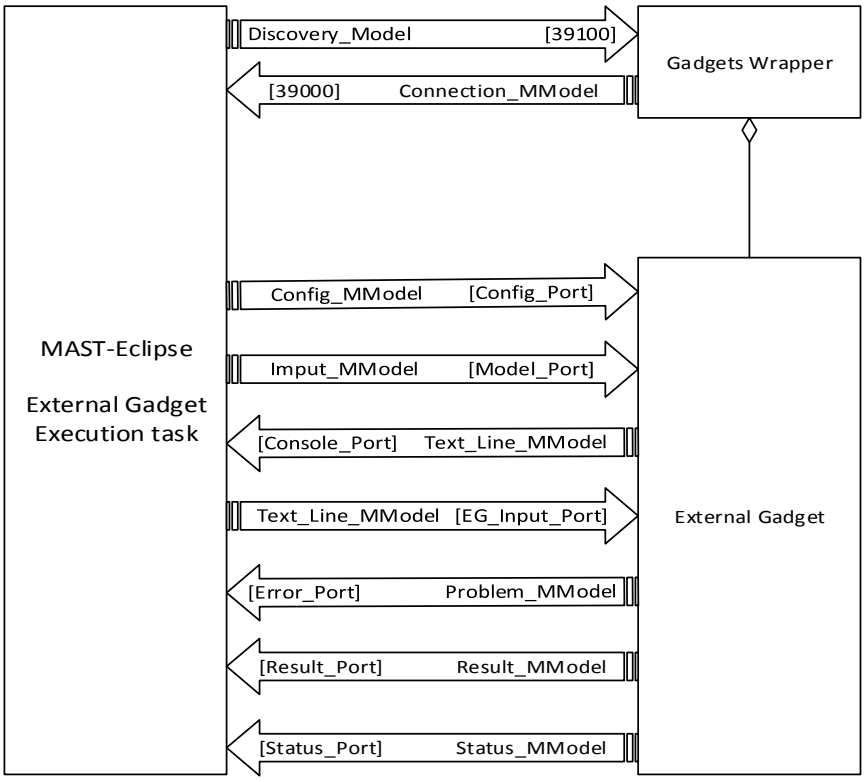

Figura 2-14. Intercambio completo de modelos entre un artefacto externo y el entorno

2.3.2 Proceso detallado de ejecución de un artefacto externo

En la Fig. 2.15 se ilustra el proceso de ejecución de un artefacto externo integrado en el entorno. El proceso puede incorporar una o varias etapas (tareas) previas a la ejecución propia del artefacto externo, las cuales dependerán del dominio considerado. En el caso de MAST, como se verá más adelante, consistirán en una serie de verificaciones de restricciones en el modelo de entrada. Las etapas posteriores tras la ejecución también dependen del dominio y de la información contenida en el modelo de resultados devuelto por el artefacto al entorno tras su ejecución.

La etapa *external gadget execution* es la que corresponde al lanzamiento del artefacto externo. Se subdivide a su vez en las subetapas que se detallan a continuación.

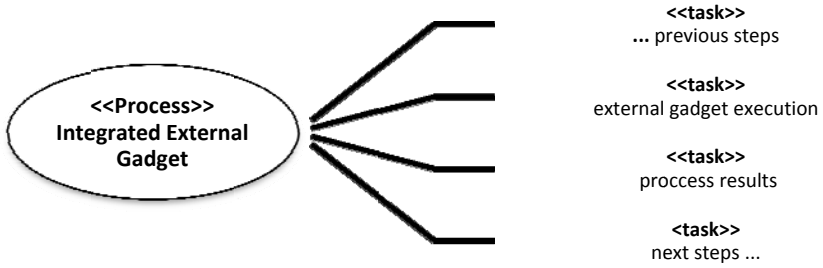

Figura 2-15. Etapas en la ejecución de un artefacto externo

1. Descubrimiento

Se explora la existencia de un *Wrapper* de una lista de nudos caracterizados por su IP, la cual representa la plataforma gestionada por el entorno. A tal fin se envía al puerto *39100* del nudo un mensaje de descubrimiento que consiste en un modelo conforme al metamodelo *Discovery_MModel*. Si el *Wrapper* se está ejecutando en el nudo, éste envía un mensaje conteniendo un modelo conforme a *Connection_MModel* con el que describe la disponibilidad para la ejecución del artefacto.

El resultado de la fase de descubrimiento puede ser uno de entre los tres siguientes:

1. No existe ningún elemento conectado al puerto *39100* en el nudo que se intenta. Ello supone que no se ha instalado el *Wrapper* en ese nudo, y por ello el lanzamiento finaliza con error.
2. La conexión se establece, pero transcurre un plazo de 1 segundo sin que el entorno reciba por el puerto *39000* un modelo (mensaje) conforme a *Connection_MModel*, lo cual representa un fallo del elemento *Wrapper*, y por tanto el lanzamiento también fracasa.
3. La conexión se establece y el mensaje *Connection_MModel* se recibe en menos de 1 segundo. En este caso el campo *Status* del mensaje es el que establece si el proceso *Discovery* se ha realizado con éxito.

Metamodelo de descubrimiento: Discovery_MModel

Corresponde al mensaje que se lanza para el Descubrimiento/Lanzamiento de la aplicación y se compone de los siguientes campos:

- *Identifier: String.* Un *Wrapper* puede gestionar varios artefactos y este campo identifica el artefacto o *gadget* que se requiere. En el caso de MAST es "*MastAnalyzerAda*".
- *Version: Natural.* Es un entero no negativo que indica la versión mínima del *gadget* que se requiere, su valor por defecto es 0.
- *Mode: Mode_Type.* Es un enumerado con valores *{DISCOVERY, EXECUTION}* que indica si sólo se desea comprobar la disponibilidad del artefacto o ejecutarlo.
- *IP: Byte[4]*. Secuencia de 4 bytes que representa la dirección IP (IPv4) del nudo que porta el entorno.
- *Connection_Port: Port_Entry[*]*. En el caso de que se solicite una ejecución se envía una lista de duplas *{Port_Name: String; Port:Natural}* que describe los puertos de conexión (de entrada o de salida) que se ofrecen al artefacto. Los identificadores *Port_Name* tienen significado dentro del artefacto y están definidos en él.

La lista de puertos con significado en *MASTAnalyzerAda* comprende:

- *Config_Port*, representa el puerto por el que se recibe el modelo de configuración del artefacto.
- *Input_Port*, representa el puerto por el que se recibe el modelo de entrada.
- *Result_Port*, representa el puerto por el que se retorna el modelo de resultados en el caso de que el artefacto los tenga que producir.
- *Console_Port*, representa el puerto por el que se envían líneas de texto hacia el marco de la consola.
- *EG_Input_Port*, por el que se reciben líneas de texto procedentes de la entrada estándar (ej: teclado).
- *Problem_Port*, representa el puerto por el que se envían notificaciones de problemas que pudiesen ocurrir durante la ejecución.
- *Status_Port*, representa el puerto por el que se envía el mensaje final de estado del artefacto.

Metamodelo de conexión: Connection_MModel

Corresponde al mensaje que devuelve el *Wrapper* para describir la disponibilidad de ejecución del artefacto en ese nudo.

Posee un único campo:

- *Status: Boolean.* Representa el resultado afirmativo o negativo de la invocación. Si la invocación del modelo de descubrimiento es *Mode=DISCOVERY*, entonces la respuesta afirmativa indica que el *Wrapper* tiene capacidad de lanzar el artefacto. Si la invocación fue *Mode=EXECUTION*, la respuesta afirmativa indica que el artefacto ha sido lanzado y se encuentra a la espera de recibir los modelos de entrada y configuración, si es que se precisan.

2. Ejecución

Si el mensaje de descubrimiento era en modo *EXECUTION*, entonces el artefacto está listo para recibir los modelos de entrada y configuración. Tras una recepción correcta de estos modelos, el artefacto se ejecutará según la configuración indicada y producirá, si es el caso, un modelo de resultados que deberá devolver al entorno.

Los modelos que se intercambian en este caso son:

- El modelo de configuración, conforme a *Config_MModel*. Representa la información que el artefacto requiere para operar sobre el modelo de entrada.
- El modelo de entrada, conforme a *Input_MModel*. Representa el modelo de entrada del artefacto. Se trata del modelo del dominio considerado.
- El modelo de resultados, conforme a *Results_MModel*. En caso de que el artefacto genere resultados, éstos se deberán enviar en un modelo de resultados conforme al metamodelo *Results_MModel*.

Estos tres metamodelos son dependientes del dominio considerado y deberán ser definidos para cada artefacto que se integre en el entorno. Tanto el entorno como el artefacto tienen conocimiento de su estructura.

Otros mensajes independientes del dominio son los que se envían en directo y en cualquier momento de la ejecución:

- Desde al artefacto al entorno: la salida por consola que produce el artefacto se envía como mensajes de tipo *Text_Line_MModel*. Para la notificación de un problema, se utiliza el metamodelo *Problem_MModel*.
- Desde el entorno al artefacto, se utilizan mensajes de tipo *Text_Line_MModel* para los datos de la entrada estándar que el entorno quiere proporcionar al artefacto.

Metamodelo para líneas de texto: Text_Line_MModel

Representa una línea de texto que es enviada desde el artefacto al marco de visualización consola del entorno. También se utiliza este metamodelo para soportar modelos que representan líneas de la entrada estándar que el artefacto necesita del entorno.

Los campos de este metamodelo son:

- *Text: EString.* Línea de texto que se transmite.
- *Is_Error: EBoolean*. Indica si el texto corresponde a un texto normal. Por defecto es *False.*

Metamodelo de problemas: Problem_MModel

Representa un problema que se ha detectado en el artefacto y se envía al marco de visualización *Problems* del entorno.

Los campos de este metamodelo son:

- *Description: EString*. Descripción textual de error orientada al operador.
- *Severity: Severity_Level.* Indica la relevancia del problema. Es un enumerado con valores *{ERROR, WARNING, ADVOCACY}.*
- *Location: EString.* Localizador textual del elemento dentro del recurso en el que se ha originado el error.
- *Resource: EString.* Localizador textual del artefacto en el que se ha generado el error.
- *Type:EString.* Descripción normalizada de la naturaleza del error.

3. Finalización

Una vez finalizada la ejecución del artefacto, éste debe enviar hacia el entorno el mensaje (modelo) final de estado, conforme al metamodelo Status_MModel.

Metamodelo Status_MModel

Define el modelo que cualquier artefacto envía cuando finaliza definitivamente su actividad. Este mensaje está constituido por dos campos:

- *Exit_Status: Finalization_Status*. Enumerado que representa el resultado global de la ejecución del artefacto. Los posibles estados de finalización son:
    - ABORTED: Se ha abortado la ejecución sin conocimiento de su causa.
    - EXIT_WITH_ERROR: La ejecución del artefacto finaliza sin éxito. Los errores producidos se envían como modelos conforme al metamodelo *Problem_MModel.*
    - SUCCESSFUL: La ejecución del artefacto finaliza con éxito.
    - EXIT_WITH_WARNING: La ejecución finaliza con éxito, pero se transfieren también mensajes de WARNING como modelos *Problem_MModel.*
    - EXIT_WITH_ADVOCACY: La ejecución finaliza con éxito, pero se transfieren también mensajes de ADVOCACY como modelos *Problem_MModel.*

  Output_File: File_Descriptor [*]. Describe los modelos que ha generado y transferido el artefacto al entorno durante su ejecución. Cada *File_Descriptor* contiene un identificador que describe la naturaleza del fichero generado desde el punto de vista del artefacto. Se excluyen de la lista los mensajes destinados a los recursos del entorno (*Console y Problems*), así como el mensaje de Status que se debe enviar siempre.

Una posible secuencia de lanzamiento de un artefacto externo se muestra en la Fig. 2-16. Los puertos son diferentes por cada modelo a intercambiar, por lo que existen agentes escuchando en un puerto específico según el metamodelo correspondiente. Tras la recepción del modelo, se procesa y se entrega al elemento activo encargado de su procesamiento.

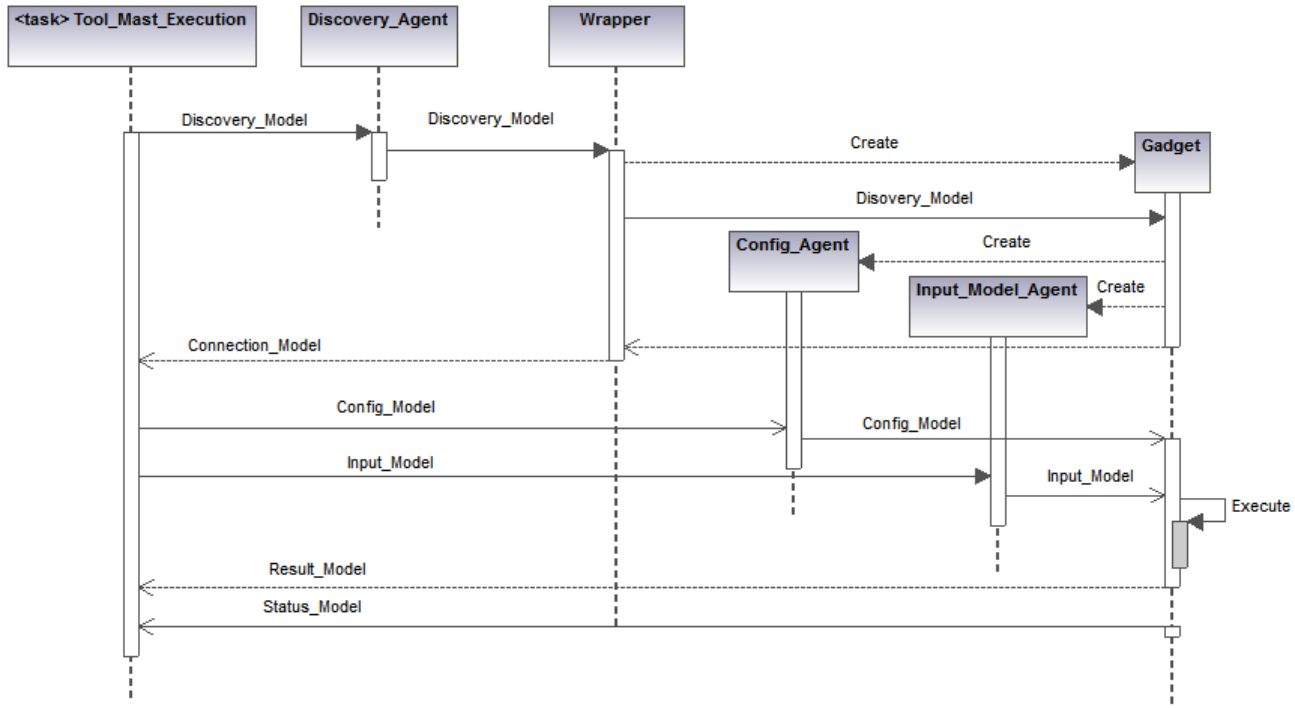

Figura 2-16. Secuencia de lanzamiento de un artefacto externo

3 Integración de las herramientas MAST 1.4 en el entorno RT-MDE

En la sección anterior se ha definido un mecanismo genérico para la integración de artefactos externos en un entorno RT-MDE. Esta sección aplica dicho mecanismo al caso particular de integración de las herramientas MAST 1.4.

Para comprender mejor el modo en que se utilizan las herramientas MAST desde el entorno, la Fig. 3-1 ilustra la herramienta de análisis de una aplicación de tiempo real tal y como se ha planteado en RT-MDE. El proceso de ejecución representa una instancia específica del proceso genérico mostrado en la Fig. 2-16 y consta de varias tareas, algunas de las cuales se llevan a cabo dentro del propio entorno, mientras que otras (en realidad, sólo una) se llevan a cabo a través de artefactos externos. Las tareas son las siguientes:

1. *Generic Mast2 Verification*. Tarea implementada dentro del propio entorno y encargada de llevar a cabo la verificación del modelo de entrada, esto es, comprobar si el modelo de entrada verifica las restricciones de integridad intrínsecas al metamodelo MAST 2.
2. *ToolMastVerification*. Tarea implementada dentro del propio entorno y encargada de llevar a cabo la verificación específica de las restricciones asociadas a la herramienta MAST elegida para el análisis.
3. *MastAnalysisExecution*. Tarea implementada a través de un artefacto externo y encargada a analizar la planificabilidad del sistema bajo desarrollo utilizando las herramientas MAST. Al artefacto externo se le ha denominado *MastAnalyzerAda*.
4. *AnalysisResultDisplay*. Tarea implementada dentro del propio entorno, que muestra los resultados del análisis en caso de que la fase anterior haya concluido sin errores.

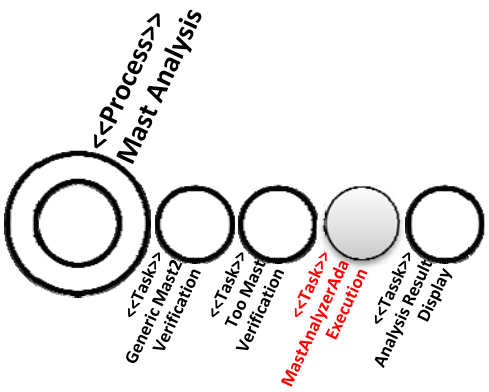

Figura 3-1 Proceso de análisis de planificabilidad en el entorno RT-MDE

Para poder llevar a cabo la invocación de *MastAnalyzerAda* se aplica el mecanismo definido en la sección anterior, cuya adaptación específica a MAST requiere:

1. Diseñar e implementar un *Wrapper*, denominado *MAST_Wrapper*, que se encargue de realizar la comunicación con el entorno RT-MDE, recibiendo y enviando los modelos de descubrimiento y conexión, respectivamente.
2. Diseñar el *Gadget*, *MastAnalyzerAda*, que se encarga de la ejecución de las herramientas MAST. Esto, a su vez, implica definir:

- El metamodelo de los modelos de entrada (Input_MModel). En este caso, el metamodelo *MAST 2.0*.
- El metamodelo de resultados (Result_MModel). En este caso, el metamodelo *MAST_Results 2.0*
- El metamodelo de configuración (Config_MModel), definiendo los datos específicos que se necesitan para configurar la ejecución de las herramientas MAST.

3. Implementar el *Gadget*. Será necesario implementar tanto las clases activas que den soporte a la interacción con el entorno (*Config_Agent, Input_Model_Agent, Keyboard_Agent, etc.)* como aquellas que den soporte a la lógica propia de MAST. Para esta última parte se reutilizará en gran medida el código de MAST 1.4, siendo necesario realizar un análisis previo del mismo, para distinguir qué elementos son reutilizables y cuáles no. Por ejemplo, MAST 1.4 aplica de forma interna tareas de verificación de restricciones, mientras que en este caso, es el propio entorno el que aplica dichas tareas de verificación, por lo que una vez que se invoca *MASTAnalyzerAda*, se tiene certeza de que el modelo de entrada es conforme y por tanto, dichas tareas de verificación no son necesarias.

La Fig. 3-2 muestra la estructura básica de soporte para ejecutar MAST como artefacto externo desde el entorno RT-MDE. La semántica y el funcionamiento de todos los elementos son los definido en la estrategia genérica, la cual fue diseñada para dar soporte a cualquier herramienta externa al entorno, siendo independiente del dominio específico. No obstante, los modelos de entrada, resultados y configuración se han especificado según la metodología de desarrollo de MAST. En la siguiente sección se describen en detalle dichos modelos de dominio así como el resto de elementos que forman la estructura de *MastAnalyzerAda* y que están orientados a dar soporte a la lógica propia de MAST.

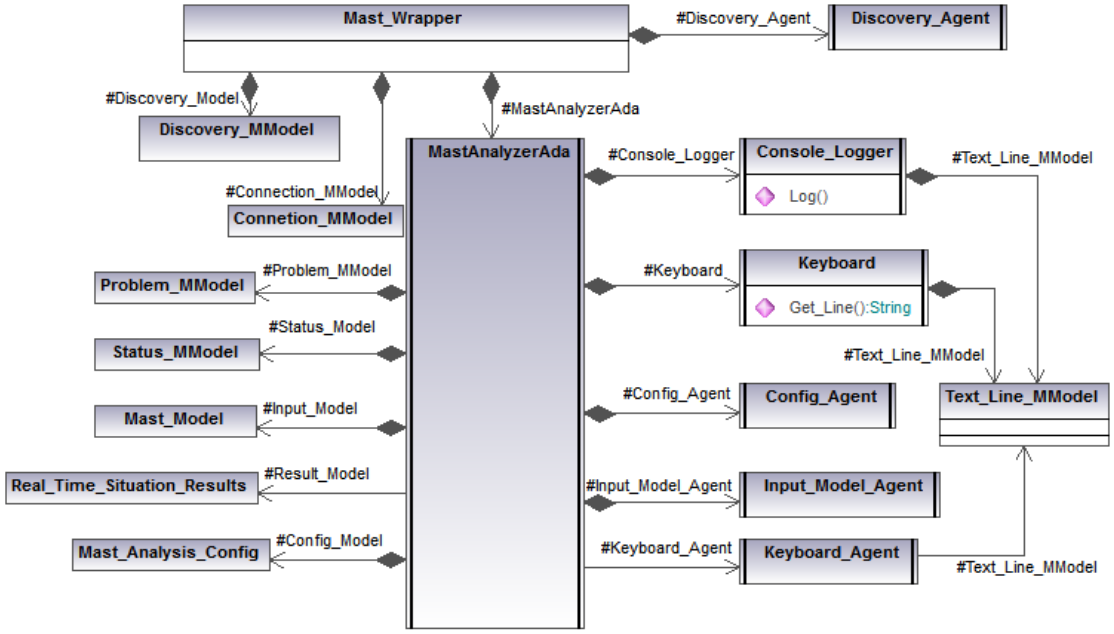

Figura 3-2 Estructura de soporte para ejecutar MAST como artefacto externo

3.1 Estructura de *MastAnalyzerAda*

La estructura completa de la clase *MastAnalyzerAda* se muestra en la Fig. 3-3. A continuación se detallan sus características.

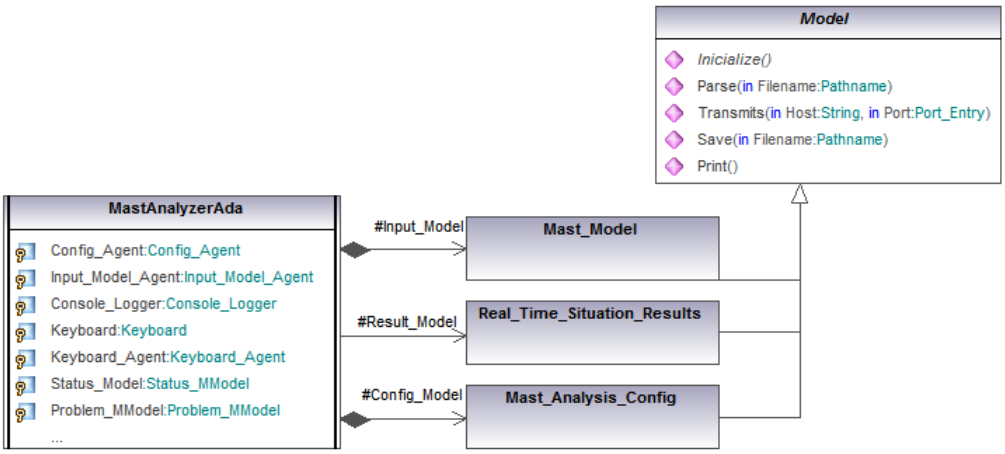

Figura 3-3. Estructura completa de *MastAnalyzerAda*

3.1.1 Clase *Model*

Se trata de una clase abstracta auxiliar que representa un modelo Ecore que puede ser utilizado desde una aplicación. Define los siguientes métodos:

- *Parse*: Permite cargar el modelo en memoria *parseando* el fichero XMI que se pasa como parámetro.
- *Print*: Imprime el modelo por consola.
- *Transmits*: Trasmite el modelo por la red en formato XMI al nodo y puerto indicados.
- *Save*: Salva el modelo en formato XMI al fichero indicado.
- *Initialize*: Inicializa el modelo.

3.1.2 Clase *Mast_Model*

Representa un modelo de entrada conforme al metamodelo MAST 2.0. En realidad correspondería a la clase Input_MModel de la estructura genérica de un *gadget*, pero se ha utilizado el nombre MAST_Model para utilizar una nomenclatura más cercana a MAST. Desde el punto de vista de *MASTAnalyzerAda*, representa el modelo de entrada para el análisis. Su estructura detallada se muestra en la Fig. 3-4. Consta de los siguientes atributos:

- *Name*: Nombre del sistema modelado.
- *Date*: Fecha de generación del modelo.
- *System_PIP_Behavior*: Comportamiento de la aplicación frente al protocolo de herencia de prioridad. Se trata de un valor enumerado que puede tomar los valores *STRICT o POSIX*.
- *Event_Queueing_Policy*: Política de cola de eventos. Enumerado cuyo valor puede ser *{FIFO, PIORITY}*.
- *Element_List*: Lista de elementos del modelo MAST 2.
- *Results*: Objeto que, tras realizar el análisis, contendrá el modelo de resultados.

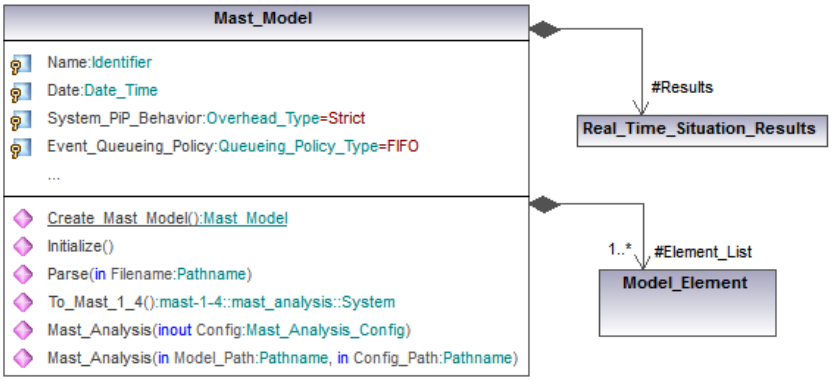

Figura 3-4 Estructura de la clase *Mast_Model*

El modelo MAST de un sistema lo forman un conjunto de elementos de modelado que se almacenan en la asociación-composición *Element_List* de la clase *MAST_Model*. La clase *Model_Element* se define como abstracta y en la Fig. 3-5 se pueden ver los tipos de elementos derivados de ella. Cada clase definida dentro de *MastAnalyzerAda* mapea de manera directa las correspondientes clases del metamodelo, por lo que su semántica y estructura se puede consultar en la descripción del metamodelo MAST 2.0 [4].

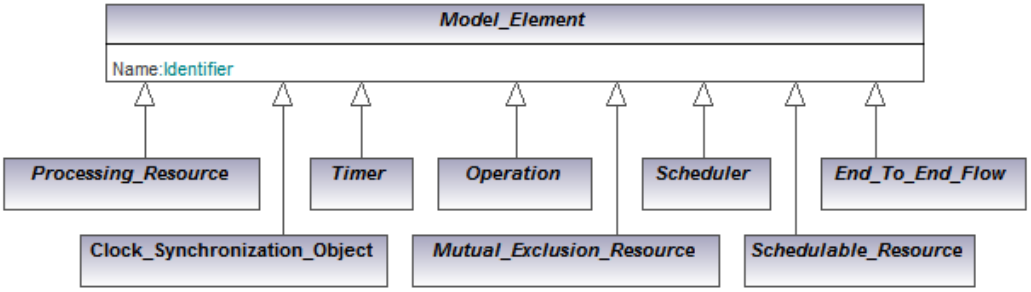

Figura 3-5 Tipos de elementos de MAST 2

Los métodos que ofrece la clase MAST_Model son:

- *To_MAST_1_4*: Transforma el modelo almacenado a un modelo compatible con MAST 1.4, por tanto, a un elemento de la clase *System* definida en el paquete MAST 1.4.
- *Initialize*: Inicializa la estructura de datos.
- *Parse*: *Parsea* el fichero XMI que se le pasa como parámetro y genera el modelo en memoria, creando los elementos correspondientes y almacenándolos en el atributo *Element_List*.
- *Mast_Analysis*: Lleva a cabo el análisis del sistema modelado. Los resultados producidos se guardan en el campo *Results* del modelo *Mast_Model*. Existen dos versiones del método. La primera lleva a cabo el análisis del modelo ya cargado en memoria. La segunda versión recibe como entrada las rutas del modelo MAST 2 y el modelo de configuración, por lo que en este caso será necesario *parsear* el modelo como paso previo al análisis.

3.1.3 Clase *Real_Time_Situation_Result*

Representa un modelo de resultados conforme al metamodelo MAST Results 2.0. Desde el punto de vista de *MASTAnalyzerAda*, representa el modelo de resultados obtenido del análisis. En realidad correspondería a la clase *Result_MModel* de la estructura genérica de un *Gadget*, pero se ha utilizado el nombre *MAST_Model* para utilizar una nomenclatura más cercana a MAST. Su estructura detallada se muestra en la Fig. 3-6.

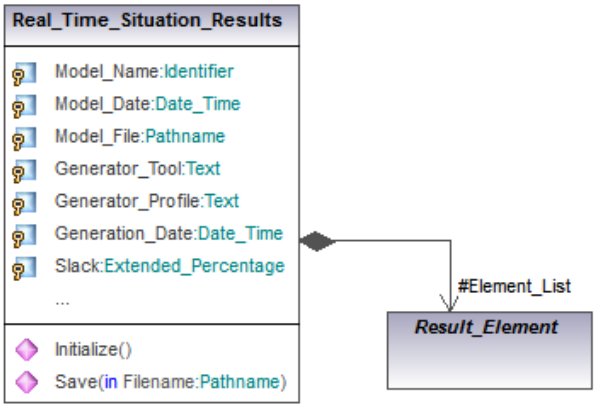

Figura 3-6 Estructura de la clase *Real_Time_Situation_Results*

Define los siguientes campos:

- *Model_Name*: Nombre del sistema modelado.
- *Model_File*: Nombre del fichero físico del modelo.
- *Model_Date*: Fecha de generación del modelo.
- *Generator_Tool*: Herramienta de análisis que generó los resultados.
- *Generator_Profile*: Cadena con las opciones de configuración utilizadas para invocar a *MastAnalyzerAda*.
- *Generation_Date*: Fecha de generación del modelo de resultados.
- *Slack*: Holgura del sistema (uno de los resultados obtenidos por las herramientas).
- *Element_List*: Elementos de resultados para procesadores, redes, operaciones, transacciones, etc. Al igual que ocurre con los elementos del modelo, la estructura y la semántica de los elementos de resultados se pueden consultar en el modelo MAST.

Como se aprecia en la Fig. 3-6, esta clase sobrescribe únicamente los métodos *Save* e *Initialize* heredados de su superclase.

3.1.4 Clase *Mast_Analysis_Config*

Representa el modelo de configuración, que almacena toda la información que el proceso de análisis requiere para poder llevarse a cabo. Su estructura se muestra en la Figura 3-7.

Dispone de los siguientes campos:

- *Analysis_Tool*: Herramienta MAST elegida para el análisis. Enumerado que puede tomar los valores *{OFFSET_BASED, OFFSET_BASED_OPTIMIZED, HOLISTIC, CLASSIC_RM, EDF_WITHIN_PRIORITIES, EDF_MONOPROCESSOR, VARYING_PRIORITIES, PARSE}*, cada uno de los cuales corresponde a una de las herramientas de análisis proporcionadas por MAST.

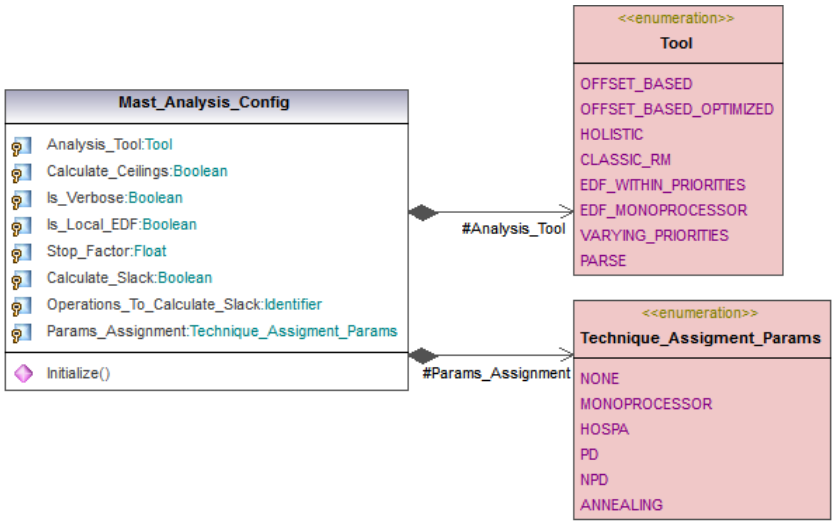

Figura 3-7 Estructura de la clase *Mast_Analysis_Config*

- *IsVerbose*: Booleano que establece la generación de información detallada para el operador.
- *Is_Local_EDF*: Establece que los tiempos establecidos en el modelo se consideren locales.
- *Stop_Factor*: Se detendrá la iteración del análisis cuando el tiempo de respuesta de una tarea exceda su plazo multiplicado por *Stop_Factor*. Por defecto es infinito.
- *Calculate_Slacks*: Se establece (o no) el cálculo de holguras *(slacks)*.
- *Operations_To_Calculate_Slacks*: Lista de operaciones para las que se desea evaluar las holguras.
- *Calculate_Ceiling*: Booleano que indica si se establece o no el cálculo de los techos de prioridad de los mútexes.
- *Params_Assignment*: Se establece y elige la herramienta de asignación de prioridades. Es un enumerado que puede tomar los valores *{NONE, MONOPROCESSOR, HOSPA, PD, NPD, ANNEALING}*. Este parámetro funciona de modo conjunto con el anterior. Si se asigna un valor a *Params_Assignment*, se utilizará la técnica seleccionada para la asignación de parámetros de planificación y de forma automática, también se asignarán techos de prioridad a los recursos compartidos.

En la Fig. 3-8 se ilustra la secuencia de lanzamiento de *MastAnalyzerAda* desde el entorno RT-MDE. Esta secuencia se corresponde con el escenario principal, donde no se produce ningún tipo de anomalía y el proceso termina de forma correcta.

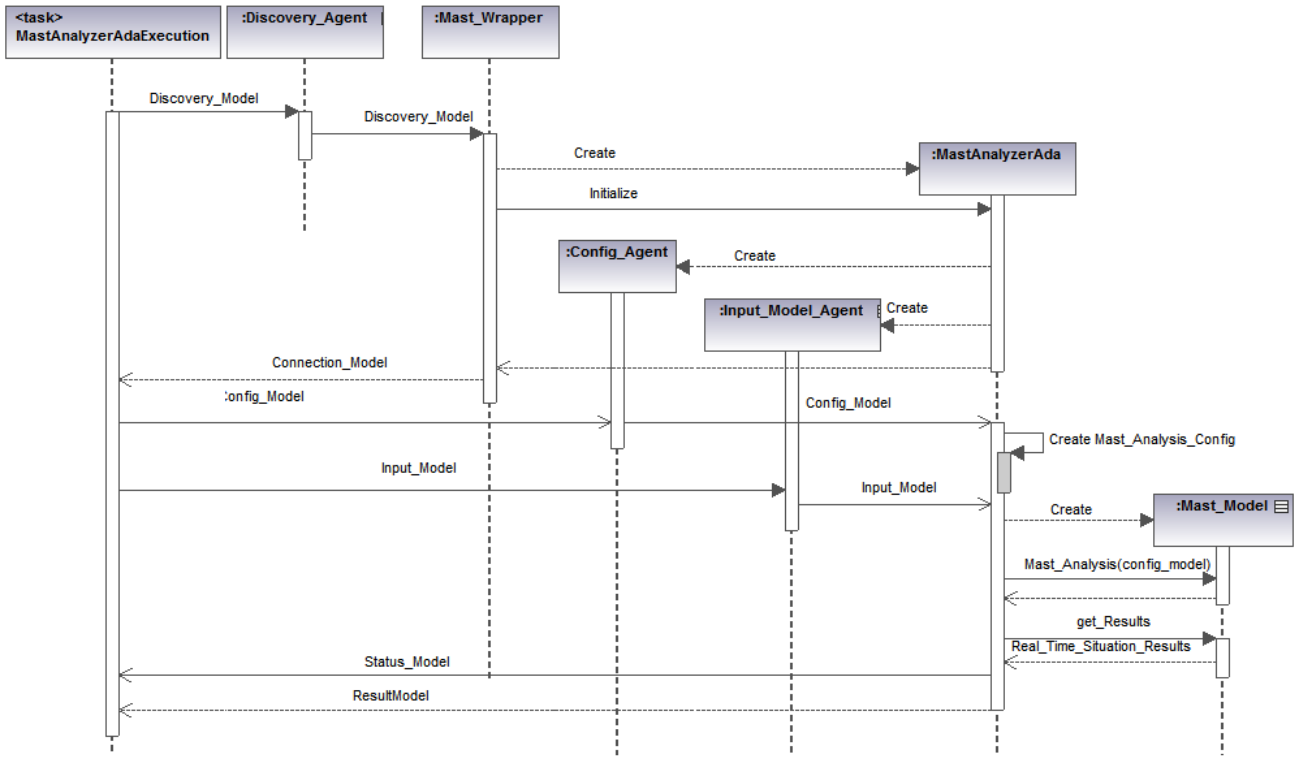

Figura 3-8. Secuencia principal de análisis con *MastAnalyzerAda* utilizado desde RT-MDE

3.1.5 Dependencias externas de *MASTAnalyzerAda*

El diagrama de paquetes de la Fig. 3-9 muestra las dependencias entre el código de *MASTAnalyzerAda* y las librerías externas utilizadas en su implementación. Cómo es de esperar, el paquete EM4Ada definido en el capítulo 2 se utiliza para realizar el procesamiento de los modelos XMI recibidos y para serializar los resultados obtenidos del análisis a modelos XMI. Así mismo, como se ha dicho al inicio de la sección, se reutiliza parte del código de MAST 1.4, en concreto, el código que da soporte a las herramientas de análisis. La Fig. 3-9 muestra la organización en carpetas del código de MAST 1.4 (paquete mast 1-4). De todas ellas, sólo ha sido necesaria la utilización de los paquetes disponibles en la carpeta *mast_analysis*, dónde se engloban las herramientas disponibles, y la carpeta *utils*, que proporciona estructura de datos auxiliares.

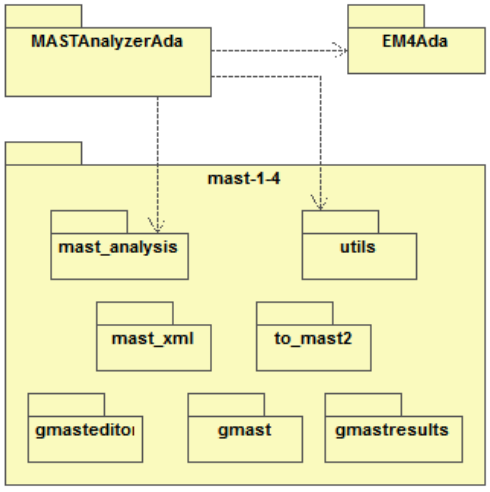

Figura 3-9. Estructura de paquetes final y dependencias

4 Conclusiones y líneas futuras

La principal aportación de este trabajo es la integración de las herramientas MAST en su versión 1.4 en el entorno RT-MDE, es decir, en un entorno integrado de desarrollo plenamente dirigido por modelos y basado en Eclipse. Para ello:

- Se ha diseñado un mecanismo para la interacción entre el entorno RT-MDE y cualquier herramienta externa escrita en un lenguaje de programación distinto de Java y/o ejecutada en una máquina virtual distinta de la del entorno. Gracias a dicho mecanismo, las herramientas pueden ser ejecutadas de manera que toda su interacción con el usuario se realiza a través de los recursos propios del entorno. Este mecanismo se ha adaptado para la integración de las herramientas MAST, pero se ha definido de manera genérica, de modo que podrá ser utilizado en el futuro para poder integrar otras herramientas, independientemente del dominio al que pertenezcan.
- Se ha diseñado una librería Ada que permite procesar la serialización a XMI de modelos conformes a metamodelos Ecore. La librería se ha utilizado para procesar y generar, respectivamente, los modelos de entrada y los modelos de resultados de las herramientas MAST en su versión 2.0. Sin embargo, la librería constituye una aportación en sí misma, ya que puede ser utilizada para procesar modelos independientemente de su metamodelo.

Como líneas de trabajo futuro que surgen a raíz de este trabajo, se pueden destacar las siguientes:

1. Desarrollo de un módulo Ada que permita procesar modelos XMI conformes a metamodelos Ecore sin requerir del usuario un conocimiento previo del metamodelo asociado. Es decir, la librería permitirá cargar en memoria tanto el modelo, como el metamodelo asociado, de manera que el usuario pueda ir accediendo a la información a través de métodos reflexivos proporcionados por la interfaz.

2. Extender la funcionalidad de la librería EM4Ada, haciéndola configurable, de manera que se pueda trabajar con identificadores explícitos. En otras palabras, poder configurar el tipo de identificadores a usar para los elementos del documento XMI, bien los basados en URI que se han utilizado en la versión actual, o los identificadores explícitos, formulados como un atributo incluido como identificador del elemento.

3. Desarrollar una tecnología para la configuración e instanciación automática de aplicaciones Ada orientadas a objeto en base a modelos EMF/Ecore. Este tipo de tecnología ya existe dentro de EMF para lenguaje Java. El objetivo es ser capaces de instanciar automáticamente aplicaciones Ada como instancias de objetos relacionados entre sí, en base a la información disponible en una instancia de modelo Ecore, que contiene:
   a) La especificación de los objetos que forman la aplicación.
   b) Los parámetros de configuración de cada objeto.
   c) Las referencias entre objetos.

5 Referencias

[1] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, pp. 25-31, 2006.

[2] M. González Harbour, J.J. Gutiérrez García, J.C. Palencia Gutiérrez, and J.M. Drake Moyano, "MAST: Modeling and Analysis Suite for Real Time Applications", Proceedings of 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, IEEE Computer Society Press, pp. 125-134, June 2001.

[3] Harbour, M. G., Gutiérrez, J. J., Medina, J. L., Palencia, J. C., Drake, J. M., Rivas, J. M., ... & Cuevas, C. MAST: Bringing Response-Time Analysis into Real-Time Systems Engineering. In *Proceedings of a conference organized in celebration of Professor Alan Burns' sixtieth birthday* (p. 42).

[4] C. Cuevas, J. M. Drake, P. López Martínez, J. J. Gutiérrez García, M. González Harbour, J. L. Medina and J. C. Palencia, "MAST 2 Metamodel," 2012. Disponible en http://mast.unican.es/jsimmast/Mast_2_0_Metamodel.pdf

[5] MAST 2: Nueva revisión del modelo de tiempo real. J.M. Drake, M. González Harbour, J.J. Gutiérrez, P. López Martínez, J. Medina y J.C. Palencia. XIII Jornadas de Tiempo Real, Granada, February 2010, ISBN: 978-84-92757-51-0, pp. 181-190

[6] Cuevas, C., Barros, L., Martínez, P. L., & Drake, J. M. (2013). Beneficios que aporta la metodología MDE a los entornos de desarrollo de sistemas de tiempo real. *Revista Iberoamericana de Automática e Informática Industrial RIAI*, *10*(2), 216-227.

[7] D. S. Steinberg, EMF, Eclipse Modeling Framework.

[8] OMG, XML Metadata Interchange (XMI) Specification. OMG document formal/05-05-01, 2000.