Universidad de Cantabria
Departamento de Ingeniería Informática y Electrónica

Metaherramientas MDE para el diseño de entornos de desarrollo de sistemas distribuidos de tiempo real

Tesis Doctoral
César Cuevas Cuesta
Santander, enero 2016

Universidad de Cantabria
Departamento de Ingeniería Informática y Electrónica

Metaherramientas MDE para el diseño de entornos de desarrollo de sistemas distribuidos de tiempo real

Memoria
presentada para optar al grado de DOCTOR por la Universidad de Cantabria por

César Cuevas Cuesta
Licenciado en Ciencias Físicas y Máster en Computación por la Universidad de Cantabria

Universidad de Cantabria

Departamento de Ingeniería Informática y Electrónica

Metaherramientas MDE para el diseño de entornos de desarrollo de sistemas distribuidos de tiempo real

Memoria
presentada para optar al grado de Doctor por la Universidad de Cantabria, dentro del programa oficial de postgrado en Ciencias, Tecnología y Computación, por el Licenciado en Ciencias Físicas y Máster en Computación por la Universidad de Cantabria
César Cuevas Cuesta

Los Directores,

Dr. José María Drake Moyano
Catedrático de Universidad y,

Dra. Patricia López Martínez
Contratado Doctor

Declaran:
Que el presente trabajo ha sido realizado en el Departamento de Ingeniería Informática y Electrónica de la Universidad de Cantabria, bajo su dirección y reúne las condiciones exigidas para la defensa de tesis doctorales.

Santander, enero de 2016

Fdo. José María Drake Moyano

Fdo. César Cuevas Cuesta

Fdo. Patricia López Martínez

En primer lugar, quiero expresar mi agradecimiento a José María Drake Moyano, director de esta Tesis Doctoral, y previamente de mi Tesis de Máster, y por encima de todo mi mentor, por haberme guiado en la elaboración de este trabajo y sobre todo por haberme dado la oportunidad de pertenecer al Grupo ISTR (anteriormente CTR), para mi mucho más que un puesto laboral.

Me habría gustado describir con palabras propias mi percepción sobre la huella que ha dejado en mí, pero tras leer el magnífico libro de Manjit Kumar ***Quantum: Einstein, Bohr, and the Great Debate about the Nature of Reality*** comprendí que alguien antes que yo, aunque en otro contexto y respecto a otra persona, había conseguido expresar exactamente lo mismo. Me estoy refiriendo a la carta que Niels Bohr (*) escribe en 1912 a su hermano Harald al poco tiempo de llegar a Manchester para incorporarse al equipo de investigación de Ernest Rutherford (**). Por tanto me limitaré a transcribir tales palabras.

> *'Rutherford is a man whom one cannot be mistaken about', Bohr wrote to Harald, 'he comes regularly to hear how things are going and talk about every little thing'. Unlike Thomson, who seemed to him unconcerned about the progress of his students, Rutherford was 'really interested in the work of all people who are around him'.*

Lo que creo que nunca sabré es si en mi caso, esta analogía Bohr – Rutherford se cumple también en el sentido contrario…

> *As soon as Bohr arrived in Manchester, Rutherford wrote to a friend: 'Bohr, a Dane, has pulled out of Cambridge and turned up here to get some experience in radioactive work'. Yet there was nothing in what Bohr had done to date to suggest that he was any different from the other eager young men in his laboratory, except the fact that he was a theorist. Rutherford held a generally low opinion of theorists and never lost an opportunity to air it. 'They play games with their symbols', he once told a colleague, 'but we turn out the real solid facts of Nature.[…] Yet he had immediately liked the 26-year-old Dane. 'Bohr's different', he would say. 'He's a football player'!*

Siendo esto último lo único que, posible y desafortunadamente, tengo en común con Bohr.

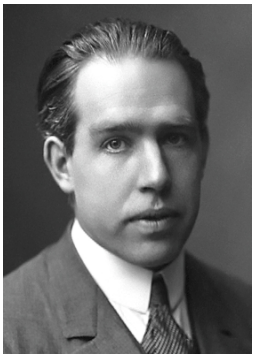
En segundo lugar, deseo dar las gracias a Patricia López Martínez, codirectora de esta Tesis Doctoral y sobre todo una gran compañera. Trabajadora infatigable, a buen seguro que al final de su carrera académica hay una cátedra con su nombre esperándola, justa recompensa a su esfuerzo. Todo lo que no sea así será una gran injusticia.

En tercer lugar, no puedo dejar de mencionar a mis compañeros del "despacho 12", aquellos con quienes un 1 de Septiembre de 2006 comencé – Álvaro, Laura, Ángela e Iñaki – y aquellos con quienes he terminado – Juan, Alex, Noelia y Jesús –. Gracias a todos.

Por supuesto, un guiño de complicidad y afecto al resto de mis compañeros de TRIS++, desde el primero hasta el último. Algunos, profesores míos en el pasado; otros, alumnos míos en el pasado…

Y por último, mis amigos, especialmente "los de Físicas", se enfadan si no me acuerdo de ellos aquí. Para no incurrir en el error social de poner nombres y dejarse a alguien en el tintero, este párrafo es anónimo, pero a buen seguro que quien se haya tomado la molestia de leer esto se sabrá incluido. ¡Va por ustedes! 

(*) **Niels Henrik David Bohr** (Copenhague, 7 de octubre de 1885 – ibíd. 18 de noviembre de 1962). Realizó contribuciones fundamentales para la comprensión de la estructura del átomo y la mecánica cuántica. Tras doctorarse en la Universidad de Copenhague en 1911, e intentar la ampliación de estudios en el Cavendish Laboratory de Cambridge con el químico J. J. Thomson, quien no mostró un gran interés, completó sus estudios en Manchester, teniendo como maestro a Ernest Rutherford, con el que estableció una duradera relación científica y amistosa. En 1922 recibió el Premio Nobel de Física por sus trabajos sobre la estructura atómica y la radiación.



(**) **Lord Ernest Rutherford**, (Brightwater, Nueva Zelanda, 30 de agosto de 1871 – Cambridge, Reino Unido, 19 de octubre de 1937). Premio Nobel de Química en 1908 por descubrir que la radiactividad iba acompañada por una desintegración de los elementos. Se le debe un modelo atómico, con el que probó la existencia del núcleo atómico, en el que se reúne toda la carga positiva y casi toda la masa del átomo. Durante la primera parte de su vida se consagró por completo a sus investigaciones y pasó la segunda mitad dedicado a la docencia y dirigiendo los Laboratorios Cavendish de Cambridge, en donde se descubrió el neutrón. Fue maestro de Niels Bohr y Robert Oppenheimer.

A mis padres

Índice de contenidos

Lista de figuras

Lista de tablas

Lista de fragmentos de código

Resumen

El objetivo principal de esta Tesis es el desarrollo de estrategias y soluciones que incentiven la adopción de la Ingeniería Software Dirigida por Modelos (MDSE) por parte de los expertos encargados del diseño de entornos de desarrollo de sistemas software en general y de sistemas distribuidos de tiempo real en particular, fomentando de esta forma su implantación como base de tales entornos. Para consolidar el enfoque adoptado, el binomio desarrollador de sistemas software (usuario de entorno) y diseñador de entornos de desarrollo se complementa con la definición de un tercer agente denominado desarrollador de infraestructuras MDSE. Su cometido principal es aligerar la responsabilidad del diseñador de entornos, de forma que no esté obligado a ser experto en tecnologías MDSE. Así, la consecución del objetivo global se ha abordado desde la perspectiva de realizar contribuciones en el campo de este tercer agente.

Se ha concebido una metodología mediante la que desarrollar herramientas genéricas que no requieran adaptación ante la evolución de los metamodelos que constituyen el ámbito conceptual de los entornos basados en MDSE. La metodología consiste en concebir la funcionalidad de cada herramienta como dependiente de un modelo que la instruye para adaptarse a cada dominio al que se desee aplicar. Más concretamente, se ha adoptado un enfoque generativo en base al cual estas herramientas genéricas están basadas en metaherramientas que bajo demanda generan la herramienta específica acorde a lo especificado por el modelo instructor. Las metaherramientas presentadas están desarrolladas a partir de la técnica HOT para generación automática de transformaciones de modelos.

Bajo el mismo objetivo central de facilitar la tarea de especificación e implementación de entornos al ingeniero software experto en su diseño, la Tesis propone una concepción genérica de entornos basados en MDSE. La propuesta considera que el operador que utiliza un entorno lleva a cabo su actividad mediante la ejecución supervisada de procesos, la cual a su vez consiste en la ejecución secuencial o iterativa de operaciones más básicas denominadas tareas. De acuerdo al espíritu MDSE, tanto procesos como tipos de tareas son formulados y suministrados en forma de modelos, siendo el entorno el encargado de interpretarlos y ejecutarlos, siempre bajo la supervisión del operador.

Se han realizado implementaciones a modo de prueba de concepto de las estrategias, metodologías y herramientas propuestas sobre el entorno MAST-2 de diseño y análisis de sistemas de tiempo real.

1 MDSE y Entornos de Desarrollo de Sistemas de Tiempo Real

Los avances en el hardware, en las redes de comunicación y en la propia Ingeniería Software inducen que día a día crezca la complejidad del software que requiere la industria y la sociedad. Para abordar este persistente incremento de la complejidad, se han de mejorar las estrategias de desarrollo de software introduciendo nuevos niveles de abstracción que faciliten la transición entre la especificación y la implementación. Asimismo, se ha de dar soporte a los patrones que representan la experiencia de diseño y cuya eficacia está comprobada.

La evolución de la Ingeniería Software es un crecimiento conducido por ciclos de crisis [1]. Cada ciclo se inicia cuando se comprueba que los métodos en uso son incapaces de abordar la complejidad del software que se necesita, avanzándose en las metodologías de desarrollo mediante la introducción de cambios relevantes que posibilitan abordar sistemas más complejos. Al cabo de un cierto tiempo se comprueba que de nuevo se ha superado la capacidad de la estrategia y se hace necesario iniciar un nuevo ciclo. Bajo esta dinámica, la velocidad con que cambian los entornos de desarrollo de sistemas software[1] hace ineficiente su soporte mediante estrategias basadas en herramientas CASE, tal y como se utilizó para la Ingeniería Software en la última década del siglo XX y como se continúa realizando en otras ingenierías.

Actualmente se considera que la Ingeniería Software Dirigida por Modelos (MDSE) es la disciplina más adecuada dentro de la Ingeniería Software para dar soporte a la rápida evolución de los entornos. Con ella, los procesos de desarrollo se conciben como una serie de pasos mediante los que los modelos relativos a la especificación y que describen el dominio del problema, van siendo refinados hasta conducir a aquellos que constituyen el dominio de la implementación, así como hasta aquellos otros que constituyen la verificación y validación de cada uno de ellos y de la correspondencia entre ambos. En MDSE, los pasos del proceso de desarrollo son considerados como simples transformaciones entre modelos.

Un modelo es básicamente un elemento de información con la representación abstracta de un sistema o de algún aspecto de un sistema. Su objetivo es proporcionar una descripción interpretable por un desarrollador humano o presentar la información con un formato que facilite su procesamiento automático mediante herramientas. Los modelos pueden representar tanto la descripción del propio problema como aspectos del sistema que está siendo desarrollado o incluso la solución o cualquiera de los estados intermedios del proceso de desarrollo. Actualmente, ya está generalizado el uso de modelos para el desarrollo de los aspectos estructurales y funcionales de los sistemas software, principalmente en base al Lenguaje de Modelado Unificado (UML) [2, 3] y sus diferentes extensiones y perfiles más específicos (SysML [4], MARTE [5], etc.). Sin embargo, la capacidad real de la MDSE se pone de manifiesto cuando se integran todos los aspectos (especificación de requisitos, formulación de restricciones, comportamiento temporal, fiabilidad, seguridad, riesgo, etc.) que concurren en el desarrollo de un sistema software.

En esta Tesis se tratan como modelos todos los elementos que se gestionan durante el ciclo de vida de un sistema software:

[1] Por brevedad, de ahora en adelante, *entornos de desarrollo* o simplemente *entornos*.

- **Modelos de desarrollo**. MDSE incentiva el uso de modelos de requisitos, de arquitectura, de implementación o de despliegue del sistema software que se utilizan, con cualquier nivel de abstracción, como parte del proceso de desarrollo.
- **Código fuente**. El código fuente de un sistema software puede ser considerado como el modelo que describe cómo se comporta el sistema cuando se ejecuta en una plataforma dada y, obviamente, es un resultado importante del proceso de desarrollo.
- **Modelos de ejecución**. Son modelos sobre aspectos de la plataforma de ejecución o del sistema software que se ejecuta y que son utilizados como base de la propia ejecución. Representan elementos de acople entre la plataforma y el software en cualquiera de los sentidos. A veces representan información de ejecución del software (configuración, despliegue, parametrización, etc.) que el supervisor de la plataforma utiliza para la monitorización y gestión de su ejecución. Otras veces, describen aspectos de la propia plataforma de ejecución (configuración interna, *middleware*, librerías, interfaz gráfica de usuario, etc.) y son interpretados por el código de la aplicación al ejecutarse.

La disciplina MDSE como base de los entornos constituye una evolución de la metodología basada en herramientas CASE, heredando de ella la experiencia que se obtuvo de su aplicación en el siglo pasado. La transición de la denominación CASE a MDSE se impuso cuando los modelos pasaron de ser utilizados como medio de documentación de algún punto de vista aislado de los sistemas software a constituirse en los elementos principales y unificadores [6] de todas las fases del ciclo de desarrollo del software (requisitos, arquitectura, diseño detallado, implementación y despliegue) y de todos los aspectos que tienen interés (funcionalidad, comportamiento, fiabilidad, seguridad, uso de recursos, riesgos, etc.). En MDSE, los modelos constituyen vistas complementarias relativas a muchos aspectos del sistema, ubicados en diferentes niveles de abstracción y generados en las diferentes etapas de su ciclo de vida.

Aunque la responsabilidad de conocer el cuerpo conceptual de MDSE y sus tecnologías para su aplicación como soporte de los entornos ha de recaer en los expertos en esta disciplina, actualmente se sigue requiriendo a aquellos expertos en Ingeniería Software encargados de proponer entornos y procesos de desarrollo de sistemas software que pretendan aplicarla el que tengan amplios conocimientos de MDSE. En esta Tesis se hace uso de la propia MDSE como medio de simplificar el empleo de sus tecnologías en el diseño y evolución de los entornos. Se han identificado recursos y procesos requeridos por estos entornos, se han abstraído y modelado sus características y se han desarrollado metaherramientas que permiten generar automáticamente los nuevos elementos de los entornos en base a modelos. Se aprovecha el hecho de que el procesado y la transformación de los modelos, que son las tareas de los entornos, también pueden ser formulados como modelos y las herramientas que los implementan generadas por medios automáticos. Con esta estrategia, la labor del experto en Ingeniería Software se reduce a formular los modelos definidos en la infraestructura, los cuales son muy próximos al ámbito del proceso en el que es experto, evitándole tener que conocer detalles de la tecnología MDSE con la que se implementan.

La Tesis explora el efecto de una infraestructura abstracta para el caso de entornos de desarrollo de software de Sistemas Distribuidos de Tiempo Real Embebidos (SDTRE)[2], que requieren tratar aspectos muy variados y heterogéneos del software.

La Tesis se plantea como una prueba de concepto, cuyo objetivo no es desarrollar una infraestructura completa, sino abordar ámbitos específicos sobre laxitud de los metamodelos, verificación, definición de vistas y extensiones, creación de modelos basados en vistas, gestión de recursos en tiempo de ejecución y organización de los recursos de los procesos. Aunque estos aspectos son sólo una parte de la infraestructura para el desarrollo de un entorno, es lo suficientemente variada para obtener experiencia de cómo estos elementos de infraestructura genérica liberan al desarrollador de entornos de ser experto en la tecnología MDSE.

1.1 Complejidad y Desarrollo de SDTREs

El desarrollo de software de SDTREs tiene una gran relevancia estratégica para la industria de un país [7], ya que alrededor del 90% de los procesadores que se fabrican se dedican a la monitorización y control de sistemas físicos (controladores de equipos, sistemas de transporte, industria aeroespacial, armamento, equipamiento médico, etc.). Esta Tesis se ha orientado al desarrollo de infraestructura MDSE genérica para dar soporte a entornos de desarrollo de estos sistemas por dos razones fundamentales:

- **El desarrollo del software de estos equipos ha sido históricamente un desafío tecnológico** que ha incentivado la evolución de la Ingeniería Software.
- **Se ha realizado en el Grupo de Investigación ISTR, con amplia experiencia en el desarrollo de estos entornos**, existiendo la motivación pertinente y requiriéndose sus resultados.

El interés que plantea el desarrollo de software de SDTREs se debe a la concurrencia en ellos de las siguientes características:

- **Funcionalidad altamente compleja**. A la complejidad inherente a cualquier sistema software, los SDTREs añaden su naturaleza reactiva y el tener que dar cobertura tanto a la falta de completitud de la especificación como a la falta de determinismo que suele existir en el comportamiento del sistema físico que controlan. La forma de abordar esta complejidad es incrementar los niveles de abstracción, tanto verticalmente, graduando el nivel de detalle de la descripción del software según se necesite en cada fase; como horizontalmente, separando los puntos de vista del sistema en base al aspecto del software que se está abordando en cada momento (especificación, arquitectura, funcionalidad, operatividad, comportamiento temporal, fiabilidad, etc.).
- **Especificación con gran riqueza y relevancia de Requisitos No Funcionales (NFRs)**. Al operar sobre sistemas que tienen su propia dinámica de evolución en el tiempo físico, la especificación del software incluye todo tipo de requisitos y restricciones de comportamiento temporal. Asimismo, cuando controlan sistemas críticos de los que dependen vidas humanas o altos costes económicos, también tienen importantes requisitos de fiabilidad, seguridad y robustez.
- **Especificidad y dependencia de la plataforma de ejecución en la que han de operar**. Históricamente, las plataformas de ejecución para sistemas embebidos han estado

[2] *Entornos SDTRE* en adelante.

caracterizadas por severas limitaciones de capacidad, memoria y consumo de energía que añadían requisitos y restricciones relativos al uso de recursos a la especificación del software. Actualmente, gracias a la evolución del hardware, se han relajado estas restricciones. Sin embargo, en contrapartida se ha generalizado el uso de plataformas distribuidas especializadas, buscando simplificar el cableado e incrementar la fiabilidad del hardware. El uso de tales plataformas conlleva la incorporación al desarrollo del software de los aspectos de configuración y despliegue.

- **Dependencia del sistema físico sobre el que operan**. En los sistemas embebidos, el software y la plataforma de ejecución interaccionan con el sistema físico controlado. Las suposiciones sobre su comportamiento y las interfaces y protocolos de interacción son aspectos que hacen compleja tanto la especificación del sistema como los patrones de implementación y los modelos para la predicción del comportamiento.
- **Presión del mercado para reducir el coste y los plazos de desarrollo**. El software de los sistemas embebidos forma parte de instrumentos y equipos muy variados que evolucionan muy rápidamente en un mercado muy competitivo. Un entorno SDTRE tiene que conciliar tres objetivos contrapuestos:
  - Funcionalidad que continuamente se enriquece y refina.
  - Reducción del tiempo y coste de acceso al mercado.
  - Satisfacción de múltiples requisitos no funcionales y de QoS.

Los entornos basados en MDSE permiten abordar estos aspectos proporcionando una metodología que trate coherente y unificadamente todos los aspectos del sistema, ofreciendo a su vez recursos comunes para extraer y procesar información, así como para reincorporar resultados en las vistas del sistema relativas a cada aspecto del mismo que se requiere en las tareas de especificación, diseño, implementación, verificación y validación.

El uso de la disciplina MDSE puede dar soporte a los entornos SDTRE, en base a que proporciona:

- **Capacidad natural para formular y gestionar los múltiples puntos de vista requeridos en el desarrollo de SDTREs**. Esto hace posible que para un sistema bajo desarrollo se extraiga, modifique y reintegre la información que corresponde al tratamiento de un aspecto específico sin cambiar ni tener en consideración el resto de información sobre el sistema.
- **Recursos para dar soporte al desarrollo de software de SDTREs frente a la variabilidad de la plataforma de ejecución y del sistema físico controlado**. El software puede ser formulado en base a modelos que describen a ambos, y a partir suyo es posible generar, mediante transformación de modelos, los correspondientes modelos de configuración y despliegue del software para que se adapte a las características de éstos.
- **Facilidad para realizar en fases tempranas el análisis y verificación de aspectos característicos de los SDTREs, como el comportamiento temporal**. Estos aspectos son influenciados por otros que se abordan a lo largo del desarrollo del sistema y por ello, su análisis y verificación debe realizarse en fases tempranas del desarrollo (antes de que el código haya sido generado). MDSE permite acometer esto en base a modelos, siendo sus resultados repercutidos sobre los modelos utilizados en fases posteriores de diseño y codificación.

- **Los modelos pueden ser aplicados a nuevas plataformas y sistemas de forma rutinaria**. Esto es posible gracias a ser un medio coherente e integrado en el proceso de desarrollo para formular los diseños genéricos y patrones de las arquitecturas e implementaciones que hacen posible el trabajo de los expertos en diseño de SDTREs.

Frente al uso común de los entornos basados en UML y sus extensiones, en los que se ofrecen recursos para el diseño arquitectural, la especificación detallada de los componentes software y la generación de código, los aspectos específicos de los SDTREs son soportados por herramientas CASE que tratan de forma independiente aquellos modelos con la información que les concierne. Ejemplo de esta visión cerrada es la especificación del perfil SPT [8] de OMG, cuyo objetivo era la normalización de los entornos para tiempo real. Fue formulada como una especificación independiente de cualquier otro proceso de desarrollo de software, por lo que quedó superada en poco tiempo ante el incremento de la complejidad de los SDTREs que se desarrollaban, en los que el análisis de planificabilidad y la configuración de planificación son sólo un aspecto más del diseño. Más recientemente, en la especificación del perfil MARTE, también de OMG y sucesor de SPT, el tiempo real aparece como una vista más del desarrollo de un sistema, integrada con otras muchas vistas (funcionalidad, gestión de recursos, consumo de energía, etc.).

La siguiente sección presenta una visión general de MDSE, antes de exponer las ventajas de su aplicación al ámbito de los SDTREs (sección 1.3).

1.2 Aplicación del modelado a la Ingeniería Software: MDSE

La MDSE [9, 10] es una disciplina dentro de la Ingeniería Software que, con el objetivo de mejorar diversos aspectos en el desarrollo del software, como la productividad, el mantenimiento o la interoperabilidad, emplea modelos con los que representar total o parcialmente los aspectos de interés en cada una de las etapas del ciclo de vida de un sistema software. Su finalidad es abordar la complejidad intrínseca de estos procesos haciendo posible que en cada fase se establezca el nivel de abstracción y automatización más conveniente.

MDSE no sólo sostiene la idea de que el empleo de modelos es esencial para la comprensión y desarrollo de software complejo, sino que proporciona los mecanismos adecuados para que esa teoría se convierta en una forma concreta de trabajar, definiendo técnicas para la creación de modelos y de sus transformaciones y combinaciones en un proceso de desarrollo de software.

1.2.1 Abstracción y modelado

La abstracción es uno de los principales procesos cognitivos aplicados por la mente humana. Consiste en encontrar aquello común en diversas observaciones para generar una representación mental de la realidad, aplicando conductas como la generalización de características específicas de entes reales, la clasificación de éstos en grupos y la agregación de entes en otros más complejos. Es una técnica ampliamente aplicada en Ciencia y Tecnología, donde a menudo se llama modelado [11]. En tal contexto, una posible definición de modelo es:

> *Un modelo es una representación simplificada o parcial de un ente real, centrada únicamente en ciertos aspectos de interés de cara a un propósito determinado.*

Por tanto, al aplicar abstracción mediante la creación de un modelo, éste desempeña una función reductora, ya que no describe la realidad en toda su extensión sino que únicamente

representa una selección relevante de las propiedades del ente modelado, para concentrarse en sus aspectos de interés. En realidad, modelar resulta inevitable. Siempre se crea un modelo mental de la realidad, ya que la mente humana tiene dificultad para pensar en cualquier cosa sin que haya sido antes modelada, con lo que, desde el punto de vista de la percepción humana, cabe decir que "todo son modelos". Por tanto, considerando tal existencia obligada de los modelos, la única opción que queda a voluntad de un diseñador es si los modelos existen sólo en su mente o si dedica esfuerzo a representarlos de forma explícita.

El propósito del modelado puede ser descriptivo (de un sistema que existe) o prescriptivo (de determinación de ámbito y detalles al estudiar un problema), siendo de suma importancia en muchos contextos científicos debido a su efectividad de descripción y a su poder de predicción. De hecho, es una técnica especialmente adecuada en los procesos de desarrollo de sistemas donde en todo momento se ha de tener en mente un modelo del objetivo. En tales procesos, los modelos permiten investigar, verificar, documentar y discutir propiedades de productos antes de que sean realmente implementados e incluso en muchos casos son empleados para automatizar directamente su implementación.

Clasificación de los modelos

Los modelos pueden clasificarse según diferentes criterios. Una manera inmediata de clasificarlos es en base al nivel de abstracción en el que se lleva a cabo el modelado. De hecho, en el diseño de sistemas de información se tienen habitualmente modelos que:

- Describen objetivos y requisitos a nivel muy abstracto, sin referirse a aspectos de implementación o ni siquiera de computación.
- Definen el comportamiento de los sistemas en términos de datos almacenados y algoritmos llevados a cabo, sin detalles técnicos.
- Definen en detalle todos los aspectos tecnológicos.

La posibilidad de modelar a diferentes niveles de abstracción conlleva la necesidad de transformaciones para convertir un modelo especificado a un nivel en otro modelo especificado a otro nivel, típicamente de menor abstracción. Esta es la esencia de la Arquitectura Dirigida por Modelos (MDA), uno de los paradigmas más importantes dentro de MDSE y que será comentado posteriormente en esta misma sección.

Otra categorización distinta tiene que ver con qué aspecto del sistema es descrito mediante el modelo. Aquellos modelos que se centran en la forma arquitectural del sistema así como en sus aspectos estáticos en términos de datos gestionados se denominan modelos estáticos o estructurales, mientras que aquellos enfocados al comportamiento del sistema, describiendo ejemplos y escenarios de evolución temporal, la colaboración entre sus componentes y los cambios en el estado interno de éstos, se conocen como modelos de comportamiento. Esta separación resalta la importancia de tener diferentes vistas sobre el mismo sistema, ya que una representación exhaustiva ha de considerar tanto aspectos estructurales como de comportamiento, preferiblemente tratados independientemente pero con interreferencias entre ellos. El modelado multivista es uno de los principios cruciales de MDSE, de forma que aplicar un enfoque MDSE a un problema típicamente conduce a construir un único modelo global que contenga toda la información sobre él pero fragmentado en múltiples vistas (o modelos parciales), cada una centrada en un aspecto determinado. Cada vista puede utilizar una

notación diferente, pero todas ellas han de ser coherentes respetando el propósito final, que es la descripción exhaustiva de un único sistema.

1.2.2 Paradigmas MDSE

En función de los objetivos que se persiguen y de las estrategias que se utilizan, se han definido diversos paradigmas de desarrollo de software bajo la denominación de MDSE. Éstos se pueden agrupar en tres categorías, de acuerdo con los escenarios más frecuentes de aplicación de las técnicas MDSE.

- **Desarrollo de Software Dirigido por Modelos** (MDSD). Es el escenario MDSE más conocido, su lado más visible. Sin embargo, el espectro de posibles aplicaciones de MDSE abarca un conjunto más amplio, incluyendo las siguientes áreas.
- **Modernización Dirigida por Modelos** (MDM) [12]. La evolución de sistemas software legados para mejorar su calidad o sus prestaciones es actualmente un problema muy abierto y una cuestión afrontada por casi todas las organizaciones. MDSE ofrece soluciones en este escenario, proponiendo abordarlo mediante extracción de representaciones de alto nivel (modelos) de tales sistemas, tomando como entrada los artefactos asociados a sus componentes (código fuente, ficheros de configuración, BBDD, documentación parcial, etc.). Cada uno de estos modelos se centra en un aspecto concreto del sistema a un nivel de abstracción diferente y se usan como primer paso del proceso de modernización del software o con otros propósitos (computación de métricas, generación de documentación, etc.).

  El principal paradigma MDSE bajo esta categoría es la Modernización Dirigida por Arquitectura (ADM) [13, 14] de OMG.
- ***Models@Runtime*** [15]. Se trata de la aplicación MDSE menos extendida y consiste en la representación mediante modelos de información del contexto de ejecución, con objetivos como la adaptación o la configuración dinámica de sistemas software en tiempo de ejecución.

Desarrollo de Software Dirigido por Modelos

Los paradigmas dentro de esta categoría poseen una orientación de Ingeniería Directa, es decir, su objetivo consiste en la creación de sistemas software en base a automatizar el proceso de desarrollo, desde los requisitos hasta la aplicación desplegada, o lo que es lo mismo, utilizando los modelos como artefactos primarios en el proceso de desarrollo de software, buscan obtener una implementación generada (semi)automáticamente a partir de ellos. Así, un proceso MDSD comienza con una representación de alto nivel de la especificación del sistema software, y a partir de ella, a través de transformaciones de modelos intermedios, se genera una implementación ejecutable. Los sucesivos modelos van generando versiones cada vez más refinadas del sistema software hasta alcanzar una versión ejecutable.

Los principales paradigmas MDSE categorizados como MDSD son la Arquitectura Dirigida por Modelos (MDA), el enfoque Factorías de Software y el Modelado Específico de Dominio (DSM).

- **Iniciativa MDA** [16-18]. Estrategia del MDSD propuesta por OMG y sustentada en el uso de sus estándares. Su objetivo es separar la funcionalidad o lógica de negocio de una aplicación respecto a su implementación en una determinada plataforma. Para ello, se basa en la especificación de modelos de alto nivel de la aplicación bajo desarrollo – Modelos

Independientes de Plataforma (PIM) –. A partir suyo, y junto a Modelos Descriptivos de Plataforma (PDM) disponibles, es posible generar Modelos Específicos de Plataforma (PSM). Finalmente, sólo restaría obtener el código de la aplicación mediante conversión de modelo a texto.

- **Factorías de Software** [19]. Es la propuesta de Microsoft y se centra en emplear artefactos (modelos, transformaciones, etc.) para construir la infraestructura necesaria para producir familias de aplicaciones en lugar de aplicaciones individuales.

- **DSM**. Es la formalización como modelado de la noción de Desarrollo Específico de Dominio (DSD), basado en la creación y utilización de Lenguajes Específicos de Dominio (DSL) [20, 21] para diseñar un sistema software utilizando conceptos y lenguajes propios de un dominio de aplicación. Un DSL, en oposición a un Lenguaje de Propósito General (GPL), está destinado a formular conceptos y comportamientos de un dominio de aplicación determinado, en base a términos y construcciones propias del dominio, lo cual facilita que usuarios no expertos en programación puedan usarlo. En el caso de DSM, el uso de DSLs se restringe a Lenguajes de Modelado Específicos de Dominio (DSML).

  Esta Tesis se ha basado extensivamente en este paradigma y por ello, en el Capítulo 2, dedicado a la representación de la información en entornos SDTREs basados en MDSE, se profundiza en él y en los DSMLs.

1.2.3 Elementos constituyentes de MDSE

Los elementos básicos de MDSE son los modelos y las transformaciones de modelos. Para la especificación de ambos se necesitan lenguajes apropiados: lenguajes de modelado y lenguajes de transformación de modelos. Los de modelado se definen a su vez mediante un lenguaje de metamodelado, propio del *framework* concreto de modelado que se emplee como base. Por su parte, los lenguajes de transformación expresan las transformaciones de forma imperativa o de forma declarativa, como un conjunto de reglas de transformación.

Lenguajes de modelado y metamodelado

Los lenguajes de modelado son lenguajes formales que permiten especificar modelos mediante una notación gráfica o textual. Se distinguen dos tipos de lenguajes de modelado: los de propósito general (GPML) y los específicos de dominio (DSML).

De forma análoga a la especificación de modelos como abstracción de entes reales, se pueden definir abstracciones de los propios modelos y representar estos últimos como instancias suyas. Tales abstracciones, que también son modelos, se denominan metamodelos.

> *Un metamodelo es un modelo que describe a un conjunto de modelos, estableciendo su estructura. Los modelos descritos por un metamodelo se dicen conformes a él o modelos-instancia.*

Así, un metamodelo puede ser interpretado constructivamente, como conjunto de reglas de producción para construir modelos o analíticamente, como conjunto de restricciones que un modelo ha de cumplir para ser conforme a él.

El metamodelado [22, 23] es la disciplina que se ocupa del estudio y creación de metamodelos y merece ser considerado uno de los fundamentos teóricos sobre los que se sustenta MDSE. De

hecho, debido a la creciente aceptación de sus paradigmas, el metamodelado se ha convertido en uno de los principales focos de atención de la comunidad software.

Al igual que se emplean lenguajes de modelado para formular modelos, para formular metamodelos se emplean lenguajes de metamodelado. Éstos son lenguajes de modelado muy particulares, que proporcionan un conjunto muy concentrado de conceptos y primitivas para construir metamodelos y que además se definen reflexivamente, con el propósito de tener una arquitectura de metamodelado finita. Los lenguajes de metamodelado más comúnmente empleados están fuertemente basados en el núcleo de los diagramas de clases UML porque metamodelar es una actividad de modelado conceptual y en MDSE se aplica un modelado conceptual OO. Así, un metamodelo se crea a partir de los conceptos de clase, atributo, asociación, etc. y se puede representar como diagrama de clases, árbol de agregación o incluso textualmente. En la práctica, los *frameworks* de metamodelado proporcionan editores para especificar metamodelos y a partir suyo permiten generar editores para definir y validar modelos-instancia.

El metamodelado es muy útil en la práctica, pudiendo ser aplicado, entre otras cosas, para formalizar la definición de lenguajes de modelado. De hecho, se considera que es la opción más apropiada para ello, pudiendo abordarse su diseño de la misma forma que se realiza el modelado de cualquier dominio de aplicación, sólo que ahora se modela un lenguaje y se trata de representar los conceptos y sus interrelaciones.

Puesto que los lenguajes de metamodelado son ante todo lenguajes de modelado, su definición también puede formalizarse mediante un metamodelo de orden superior. Éste tendría rango de metametamodelo, esto es, un modelo para representar metamodelos válidos y se define mediante el propio lenguaje de metamodelado al que formaliza, es decir, los lenguajes de metamodelado se definen reflexivamente y gracias a ello se consigue una arquitectura de metamodelado finita.

En MDSE, a pesar de la estandarización de MOF [24] por parte de OMG, el lenguaje de metamodelado estándar *de facto* es Ecore, formalizado mediante el metametamodelo de igual nombre. Constituye la parte nuclear de toda la infraestructura de modelado presente y futura de Eclipse. La sección 1.4.2 profundiza en ella y en Ecore.

Arquitectura de metamodelado de 3+1 capas

En teoría, se pueden definir infinitos niveles de metamodelado. Sin embargo, en la práctica se ha demostrado que los metametamodelos pueden ser definidos a partir de sí mismos, por lo que no tiene sentido ir más allá de este nivel de abstracción. Para establecer la relación entre modelos y sus metamodelos resulta muy útil la noción esquemática de *arquitectura o pila de metamodelado de 3+1 capas*. En ella, cada nivel se denota M_i (terminología OMG) y se cumple que un modelo en M_i es conforme a un modelo en M_{i+1}. La Figura 1.1 muestra esta disposición, donde pueden identificarse las siguientes relaciones entre niveles: *representado por*, *conforme a* e *instancia de*.

Para comprender más ampliamente las relaciones recogidas en la Arquitectura de 3+1 capas, es útil establecer paralelismos entre el Modelado y otros paradigmas de representación de la información. En el capítulo 2 se abordan tales analogías.

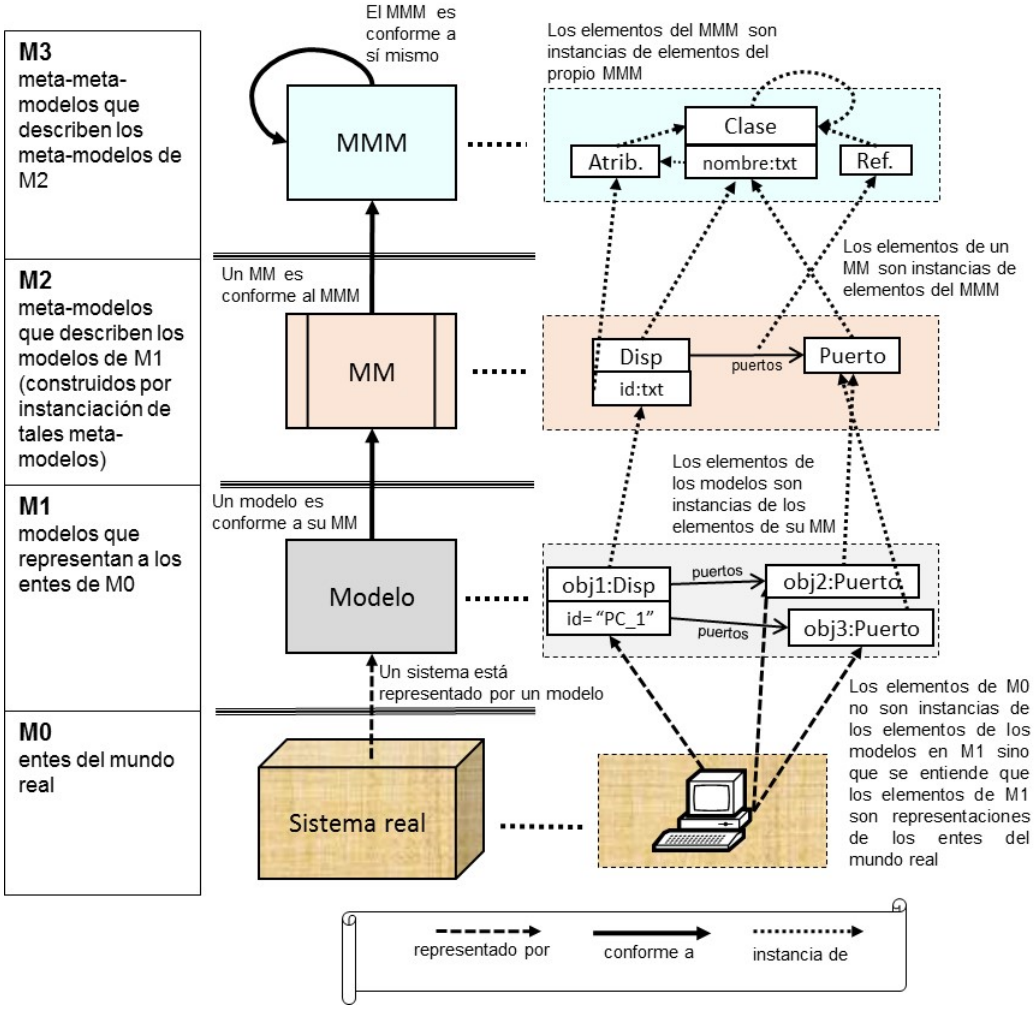


Figura 1.1 – Arquitectura de metamodelado de 3+1 capas

Transformaciones de modelos

Las transformaciones de modelos son cruciales en MDSE [25]. Los modelos no son entidades aisladas y/o estáticas, sino que experimentan diversidad de operaciones (fusión, alineamiento, refactorización, refinamiento, traducción a otros lenguajes o representaciones, etc.) que generalmente se implementan en base a transformaciones de modelos.

Existen varios criterios para caracterizar y categorizar las transformaciones de modelos [26-28].

- **En función de la naturaleza de las entradas y salidas**, puede hablarse de transformaciones de modelo a modelo (M2M), de modelo a texto (M2T) y de texto a modelo (T2M). Por ejemplo, los generadores de código se pueden considerar transformaciones M2T, y los procesos de ingeniería inversa que permiten extraer modelos a partir del código son casos de transformaciones T2M.

  Dentro de M2M se puede distinguir entre transformaciones exógenas, en las que los modelos de entrada y salida son conformes a metamodelos diferentes y endógenas, que tienen lugar entre modelos conformes a un mismo metamodelo, esto es, expresados mediante el mismo lenguaje.

- **Atendiendo al nº de modelos de entrada y/o salida** se puede establecer una clasificación muy trivial. En la mayoría de los casos, las transformaciones son 1-1, pero no son extrañas las situaciones donde se requieren transformaciones 1–n, m–1 o incluso m–n.
- **En función de la capa de metamodelado donde residan los modelos**, pueden tenerse transformaciones horizontales (M2M o incluso MM2MM), promocionadoras (M2MM) o degradantes (MM2M).

La Figura 1.2 muestra el clásico esquema de una transformación M2M convencional 1-1. Como puede observarse, aunque una transformación esencialmente supone una correspondencia entre modelos, se define sobre los correspondientes metamodelos, haciendo referencia en su formulación a conceptos y términos suyos, y se aplica sobre modelos conformes a ellos para procesar la información contenida.

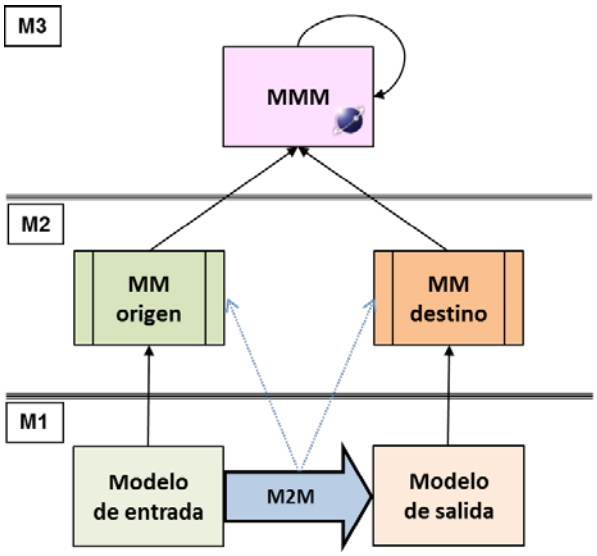


Figura 1.2 – Esquema básico de transformación M2M (exógena y 1-1)

Aunque en principio se podría utilizar cualquier lenguaje de programación para desarrollar una transformación de modelos, un lenguaje específicamente diseñado para la definición de transformaciones facilita la tarea. En los últimos años han surgido varias herramientas o lenguajes para el desarrollo de transformaciones, que adoptan diferentes aproximaciones o estilos. Los más interesantes son el estilo declarativo (o relacional), el imperativo y el híbrido.

Los lenguajes más comúnmente utilizados son:

- **QVT** (*Query View Transformations*) [29, 30] de OMG.
  Se desglosa en *QVT Relations* y *QVT Operational Mappings*, ejemplos por excelencia de las aproximaciones declarativa e imperativa, respectivamente.
- **ATL** (*ATLAS Transformation Language*) [31-33]. Es el ejemplo paradigmático de la aproximación híbrida. Su madurez dentro de Eclipse y su riqueza de recursos, tanto documentales como en cuanto a comunidad de usuarios, lo han convertido en el estándar *de facto* en cuanto al desarrollo de transformaciones de modelos, y es el lenguaje aquí utilizado.
- **ETL** (*Epsilon Transformation Language*) [34].

En definitiva, las transformaciones de modelos son herramienta clave para soportar un proceso *model-driven* y su automatización es la única forma de convertir en realidad las ventajas teóricas

que aporta MDSE, pues el esfuerzo que requeriría implementar manualmente cada transformación conduciría a descartar el uso de este tipo de proceso.

1.2.4 Ventajas de MDSE

La última década ha presenciado la progresiva adopción, aplicación y consolidación de MDSE como paradigma de desarrollo de software. Evidentemente, esto se debe a los beneficios que se derivan de ella, como la elevación de los niveles de abstracción y automatización en la construcción de software, con el consiguiente aumento de la productividad y mejora de la calidad o las posibilidades de descripción de múltiples aspectos (modelado multivista), de empleo e integración de herramientas genéricas reutilizables o metaherramientas y de independencia de plataforma. Todos ellos y muchos más aparecen en la bibliografía relacionada. En la siguiente sección se presenta una visión de tales ventajas desde el punto de vista del desarrollo de SDTREs.

1.3 Aplicación de MDSE a los entornos SDTRE

1.3.1 Conexión entre el ámbito de los SDTREs y el Modelado

El diseño de SDTREs ha sido siempre un proceso basado en modelos como consecuencia de la existencia en su especificación de diferentes tipos de requisitos no funcionales y de la frecuente imposibilidad de llevar a cabo verificaciones sobre el sistema implementado final. Por ejemplo, cuando se analiza o diseña el comportamiento temporal de este tipo de sistemas, la información requerida para aplicar análisis de planificabilidad y configuración de planificación se formula a través de modelos de comportamiento temporal (modelos TR) del SDTRE que se está analizando o diseñando. Lo mismo ocurre cuando se analizan otros aspectos específicos de estos sistemas, como fiabilidad, seguridad, etc., los cuales exigen informaciones específicas adicionales que también se formulan como modelos.

Sin embargo, a pesar del empleo de modelos, en los entornos SDTRE no es habitual utilizar estrategias MDSE. En ellos, la descripción de los diferentes aspectos de interés se realiza mediante modelos independientes y los resultados de los análisis se utilizan directamente en el diseño del código que implementa el sistema o, en los casos más complejos, se transfieren como parámetros de configuración en el lanzamiento de su ejecución.

A lo largo de su progresivo asentamiento, y debido a los beneficios que comporta, MDSE se ha ido aplicando con gran éxito en diferentes tareas de la Ingeniería Software, incluyendo el desarrollo de Sistemas Distribuidos y Embebidos de Tiempo Real (DRES). De hecho, la bibliografía relacionada muestra que la aplicación formal de técnicas MDSE a las diferentes áreas de este contexto se ha ido consolidando durante la última década [35-42].

1.3.2 Ventajas de la adopción de MDSE

El empleo de MDSE en los entornos SDTRE proporciona beneficios tanto para el desarrollador de SDTREs como para el desarrollador de herramientas, en base a que ambos pueden disponer de los modelos adecuados a su tarea. Trabajar bajo el paradigma *model-driven* conlleva que toda la información correspondiente al SDTRE o herramienta bajo desarrollo se formula como modelos cuya estructura se encuentra formalizada a través del correspondiente metamodelo que recoge los conceptos del dominio de trabajo. La utilización de un único metametamodelo,

propio del *framework* MDSE concreto sobre el que se construye el entorno, facilita el mantenimiento de las referencias entre los modelos así como sus transformaciones.

Por otro lado, además de hacer posible que en un entorno se puedan proporcionar modelos adaptados a cada punto de vista o nivel de abstracción, existe un conjunto de tareas básicas de gestión de la información (introducción, visualización, verificación de corrección, análisis de coherencia, integración, partición, almacenamiento y recuperación, etc.) a las que cualquier infraestructura MDSE proporciona soporte. Esto se debe a que pueden ser formuladas en base al metametamodelo, lo que hace que las herramientas que las realizan tengan capacidad de operar sobre cualquier modelo, con independencia de su metamodelo, y por ello son proporcionadas como parte de la infraestructura. De nuevo, esto implica beneficios para ambos tipos de desarrolladores.

Beneficios en el desarrollo de SDTREs

En general, para el desarrollador de SDTREs se incrementa la facilidad de uso de los entornos, gracias a los siguientes beneficios:

- **Adecuación del nivel de abstracción en la descripción del sistema a la tarea que se aborda**. Al centrar la atención y el razonamiento en los modelos (y no en los datos, código fuente o algoritmos, como era tradicional) se consigue mucha más proximidad a la forma de pensar humana y con ello se facilitan los procesos de desarrollo, pues se posibilita una mejor comprensión y gestión de los mismos.
- **Independencia de la plataforma**. Razonar a nivel del dominio de trabajo, con independencia de la plataforma del entorno de desarrollo que se utilice o de la plataforma en que se vaya a ejecutar la aplicación resultante, facilita la interoperatividad entre entornos y la reusabilidad de los resultados.
- **Gestión de múltiples aspectos**. El empleo de modelos para describir un sistema es clave en los entornos SDTRE, pues permite gestionar de forma integrada en un mismo entorno tantos aspectos del sistema bajo desarrollo como se necesite, de forma independiente y con la riqueza de detalle deseada. Cada aspecto se describe con tantos modelos como sea necesario, cada uno constituyendo una vista del aspecto en cuestión, aunque como toda la información corresponde al mismo sistema, el entorno debe mantener la coherencia y correspondencias necesarias. De esta manera, en función del paradigma de desarrollo que se utilice y de la fase del proceso de desarrollo, el entorno ofrece una vista especializada del sistema, la cual presenta de una forma precisa y coherente la información sobre la que se está trabajando.

  Por ejemplo, esto permite considerar un entorno dedicado únicamente al diseño de TR como una vista especializada que se integra coherentemente en un entorno más general, el cual también soporta las restantes fases del desarrollo del sistema.
- **Herramientas reutilizables de gestión básica de la información**. Su uso generalizado en cualquier ámbito de interacción con el usuario le otorga una gran facilidad de uso, pues éste reconoce en el entorno mecanismos que le son familiares.

Beneficios en el desarrollo de herramientas

La construcción y el mantenimiento de los entornos SDTRE se simplifican en caso de estar basados en infraestructura MDSE, ya que se obtienen los siguientes beneficios:

- **Se facilita el diseño, mantenimiento e integración de herramientas**. A través de los mecanismos de gestión y transformación de modelos con los que está dotada la infraestructura de base, es posible que cada herramienta reciba únicamente la información que realmente necesita y estructurada de modo adecuado. Asimismo, éstas pueden generar sus resultados de la forma que les sea natural, ya que el entorno los sabe adaptar y gestionar.
- **Herramientas reutilizables de gestión básica de la información**. Simplifican el desarrollo de herramientas específicas, ya que pueden ser integradas en éstas últimas para llevar a cabo gran parte de las etapas de gestión de la información. Así, las herramientas específicas del entorno simplemente las usan y no deben implementar su funcionalidad sino sólo su propio algoritmo de procesamiento específico, resultando por ello más sencillas.
- **Desarrollo de metaherramientas**. Las transformaciones de modelos pueden ser ellas mismas especificadas formalmente como un modelo. Esto facilita el desarrollo de metaherramientas, esto es, herramientas cuya función es generar nuevas herramientas más específicas respecto al sistema que se considera.
  El capítulo 3 se dedica en profundidad al desarrollo de metaherramientas.

Herramientas reutilizables genéricas de gestión básica de la información

A continuación se describen ejemplos de recursos típicamente disponibles en una infraestructura MDSE y de tareas que son fácilmente realizables mediante ella, y que por tanto simplifican el uso del entorno tanto a desarrolladores de aplicaciones como a desarrolladores de herramientas:

- Entorno gráfico amigable.
- Introducción y modificación de datos.
- Presentación de resultados.
- Almacenamiento y recuperación de la información.
- Validación de la información.
- Adecuación de la información y recuperación de los resultados.
- Importación y exportación de la información.
- Integración de herramientas externas.

1.3.3 Caracterización de los entornos SDTRE a la luz de MDSE

La elección de MDSE como base de entornos SDTRE, proporciona una visión más refinada de los objetivos y requisitos planteados sobre éstos.

En la concepción de entornos SDTRE en el marco de esta Tesis:

- **Toda la información gestionada en un entorno se formula y organiza mediante modelos conformes a metamodelos existentes en él**. La definición en base al *framework* MDSE que

da soporte al entorno facilita su gestión (introducción, presentación, almacenamiento, transformación, mantenimiento de la coherencia, etc.) mediante estrategias y herramientas comunes.

- **La integración de nuevos aspectos del sistema o de nuevas herramientas se basa en la incorporación de nuevos metamodelos y en la formulación de modelos de integración**, tratando simultáneamente de reducir la necesidad de generar nuevo código.
- **Se fomenta la incorporación de metaherramientas**, capaces de operar sobre modelos independientemente de los metamodelos respecto a los que son conformes. Con ello se consigue reducir el coste de integrar nuevos aspectos del sistema.

En esta Tesis se dedica el capítulo 3 al desarrollo de metaherramientas que en base a los metamodelos de la información de entrada y salida, generan automáticamente las herramientas concretas correspondientes a cada caso.

1.4 Trabajos relacionados

1.4.1 Plataformas de desarrollo para creación de entornos DSM

El empleo de DSMLs implica ventajas, especialmente permitir centrarse en los conceptos del dominio considerado, pero también conlleva ciertas desventajas, siendo la principal el esfuerzo necesario tanto para diseñar los propios DSMLs como para construir el correspondiente entorno de herramientas de soporte (herramientas CASE), a la medida del lenguaje en cuestión. La codificación manual desde cero de un entorno así es una tarea compleja que requiere una considerable cantidad de tiempo y esfuerzo, demasiado lenta para sustentar el mantenimiento y la evolución del ritmo del desarrollo de lenguajes de modelado. La solución pasa por disponer de herramientas genéricas que puedan ser adaptadas para soportar cualquier lenguaje de modelado. A este fin, tradicionalmente se han considerado dos posibles enfoques:

a. Diseño y construcción de herramientas de modelado de forma modular, de tal manera que para modificar la parte relativa a un cierto lenguaje de modelado se requiera un esfuerzo de codificación mínimo.
b. Capturar la caracterización de la herramienta CASE requerida para a continuación generarla automáticamente.

Durante la década de los 90, el primer enfoque fue principalmente adoptado por la industria para, dado un código ya existente, desarrollar nuevas versiones de herramientas para nuevos lenguajes de modelado. Sin embargo, aunque la reducción del coste para construir una herramienta de modelado era significativa, este enfoque se mostró insuficiente para satisfacer las necesidades del usuario, puesto que únicamente el vendedor podía hacer los cambios – cuyo coste era alto – y el tiempo desde que se requerían tales cambios hasta obtener la herramienta modificada era excesivamente largo.

El punto de partida para aplicar el segundo enfoque son las plataformas de desarrollo conocidas como herramientas metaCASE, que en general proporcionan componentes genéricos de herramientas CASE, los cuales pueden instanciarse de manera específica para dar lugar a herramientas CASE particulares. Típicamente, estas plataformas proporcionan un lenguaje de metamodelado así como asistencia para creación de metamodelos y generación de

implementaciones. En un principio, esta estrategia produjo prometedores prototipos en proyectos de investigación e incluso algunos productos comerciales, y aunque estas primeras metaherramientas no fueron ampliamente usadas, son las precursoras de la estrategia en que se basan los entornos DSM actuales.

Primeras herramientas metaCASE

El empleo de las primeras herramientas metaCASE mostraba que tanto la configuración de las herramientas para soportar un nuevo lenguaje de modelado (proceso de metamodelado) como el soporte para el lenguaje de modelado en la herramienta configurada podían mejorarse. De hecho, estas metaherramientas de los 90 tendían a separar la parte de metamodelado de la parte de modelado. El lenguaje de modelado se describía textualmente para posteriormente ser compilado a otro formato que alimentaba a la herramienta configurable, la cual a partir de ese momento ya soportaba el nuevo lenguaje. Algunas herramientas avanzadas (como MetaEdit y ToolBuilder) incluso proporcionaban parcialmente soporte gráfico para metamodelar. En ellas, la construcción gráfica del metamodelo se realizaba con la misma funcionalidad que la herramienta de modelado resultante, pero usando un lenguaje especial para metamodelado gráfico. Así, el metamodelo resultante tenía que ser transformado primero a lenguaje textual y después ser compilado. Esta separación de metamodelado y modelado tenía una desventaja importante: no era posible testear interactivamente los resultados del metamodelado de manera inmediata, debido al largo ciclo *transformar-compilar-enlazar-ejecutar* para pasar de metamodelado a modelado. Algunas de estas herramientas metaCASE usadas en los 90 en proyectos a escala industrial y precursoras de las plataformas DSM actuales son:

- DoME
- TBK + ToolBuilder
- MetaEdit
- FUJABA

A continuación se presentan brevemente:

- **Domain Modeling Environment (DoME)**. Herramienta metaCASE creada por Honeywell Labs para, en principio, uso interno en sus propios proyectos. Enseguida quedó de manifiesto que debería ser posible generar herramientas (en aquel caso editores) a partir de una especificación y a este enfoque generativo se le llamó MetaDoME [43]. Los siguientes desarrollos de ProtoDoME y CyberDOME supusieron el salto de generación a interpretación directa de la especificación de herramienta.

Una descripción completa de DoME puede encontrarse en [44]. DoME usaba un lenguaje gráfico de metamodelado que cubría los principales conceptos de esta disciplina (gráfico, nodo, puerto y conexión) así como conceptos más especializados relacionados con componentes e interfaces y también restricciones y comportamiento gráfico. Debido a su codificación en Smalltalk se le acusaba de presentar una UI no estándar, además de bastante austera, y el empleo de ficheros para modelos se veía como un factor limitante para la adopción de DoME por equipos numerosos, ya que sólo un usuario podía editar un modelo en un momento dado. Además, el soporte para descomponer un modelo en fragmentos editables por separado resultaba pobre. Posteriormente, DoME fue liberado como código abierto, aunque tal código ha permanecido sin cambios desde el año 2000.

- **Tool Builders Kit (TBK) + ToolBuilder**. El *framework* TBK junto al sistema metaCASE ToolBuilder [45] tiene su origen en la Universidad de Sunderland. Utilizaba Entidad-Relación (ER) como lenguaje de metamodelado, enriquecido con algunas restricciones y la habilidad de poder definir atributos derivados. A pesar de que ToolBuilder no tuvo éxito como herramienta metaCASE, algunas herramientas de modelado basadas en ella tuvieron más suerte y además, un precursor de TBK se usó para construir el conjunto de herramientas HOOD [46] usado en el proyecto Eurofighter.

- **MetaEdit**. MetaEdit [47] fue una herramienta genérica de modelado que daba soporte a lenguajes de modelado concretos en base a utilizar metamodelos en forma de ficheros binarios, empleando un lenguaje de metamodelado basado en el lenguaje OPRR (Objeto – Propiedad – Relación – Rol) [48]. Aparte de la posibilidad de construir metamodelos de forma puramente textual, MetaEdit incluía un lenguaje gráfico que contenía los conceptos OPRR con el cual los usuarios podían definir gráficamente sus propios lenguajes de modelado. El resultado era compilado para generar el fichero binario correspondiente, el cual pasaba a gobernar la herramienta genérica de modelado. La facilidad y naturaleza incremental del metamodelado fueron reconocidas como las principales ventajas de MetaEdit y, aunque también tenía varias limitaciones, (principalmente en su funcionalidad como herramienta de modelado) MetaEdit se mostró hasta cierto punto exitosa como herramienta metaCASE, con miles de usuarios. Su evolución fue MetaEdit+ (ver más adelante).

- **FUJABA (*From UML to Java and Back Again*)**. Aunque el entorno de *software* libre FUJABA inicialmente no constituye una herramienta metaCASE, sino una simple herramienta CASE monolítica destinada principalmente a dar soporte a ingeniería *roundtrip* para UML y Java, se incluye aquí una breve reseña debido a su carácter antecesor de *Fujaba Tool Suite* (ver más adelante). El proyecto FUJABA surgió a finales de 1997 en el Grupo de Ingeniería de Software de la Universidad de Paderborn y fue progresivamente evolucionando hasta ser rediseñado en 2002, convirtiéndose en *Fujaba Tool Suite*. Una visión global del entorno FUJABA inicial, sus conceptos y características principales, puede encontrarse en [49] y [50]. Básicamente, la distinción principal frente a otras herramientas UML contemporáneas radicaba en que FUJABA extendía las típicas capacidades de generación de código soportadas por aquellas (a partir de diagramas de clases, generación de esqueletos de clases y declaraciones de métodos sin cuerpo) generando código a partir de diagramas UML conductuales, como por ejemplo diagramas de colaboración. Además, FUJABA no se limitaba a ingeniería directa sino que también soportaba la extracción de diagramas UML a partir de código Java, logrado razonablemente bien para diagramas de clases y hasta cierto punto para la combinación de diagramas de actividad y de colaboración.

Plataformas actuales

Actualmente existen plataformas muy consolidadas para creación de soluciones DSM, normalmente procedentes de investigación académica, como por ejemplo:

- Eclipse Modeling Project (EMP)
- MetaEdit+
- Generic Modeling Environment (GME)
- Microsoft MS DSL Tools
- Fujaba Tool Suite

- MOFLON
- IBM Rational Software Architect (RSA)
- Obeo Designer

A continuación se presentan brevemente:

- ***Eclipse Modeling Project* (EMP)**. Dentro del conjunto de proyectos (de primer nivel) que vertebran Eclipse, el encargado de centralizar y coordinar las actividades y desarrollos MDSE es EMP[3] [51]. En esencia, se trata de una colección de subproyectos relacionados con tecnologías MDSE y modelado en general a los que contribuye un amplio abanico de organizaciones, tanto empresas comerciales como instituciones académicas.

El proyecto fundamental dentro de esa colección es *Eclipse Modeling Framework* [4] (EMF) [52], el cual cubre el desarrollo de sintaxis abstractas para lenguajes de modelado. Constituye la base de EMP, de forma que el resto de subproyectos pueden considerarse satélites suyos. En realidad EMF es anterior a EMP, habiendo formado parte de la plataforma Eclipse desde su origen.

Empleando el metamodelado como estrategia para formalizar la sintaxis abstracta de un lenguaje de modelado (tendencia más ampliamente practicada en la actualidad), EMF propone un lenguaje de metamodelado denominado Ecore, piedra angular de toda la infraestructura de modelado presente y futura basada en Eclipse. Por coherencia con tal estrategia, EMF formaliza Ecore mediante un metametamodelo, de igual nombre. La subsección 1.4.2 profundiza en EMP.

- ***MetaEdit+* (1995 –)**. Herramienta metaCASE para construcción y uso de soluciones DSM. Su desarrollo arrancó inicialmente en el grupo de investigación *MetaPHOR* [5] (Universidad de Jyväskylä) pero progresivamente se trasladó a la compañía MetaCase[6]. *MetaEdit+* desciende de *MetaEdit* en cuanto a conceptos y equipo humano pero supuso un comienzo nuevo en términos de código, intentando rectificar ciertas decisiones arquitecturales en *MetaEdit* que habían demostrado restringir su escalabilidad y eficacia.

Una visión general y concisa del lugar de *MetaEdit+* en el escenario DSM puede encontrarse en [53], mientras que en [54] y más brevemente en [55] se describen algunas características avanzadas requeridas en las herramientas de construcción de soluciones DSM. Se trata de funcionalidades relativas a necesidades detectadas en proyectos industriales y todas ellas disponibles en *MetaEdit+*.

Básicamente, *MetaEdit+* proporciona la funcionalidad estándar de una herramienta CASE, incluyendo editores gráficos, gestión de datos de diseño e integración con otras herramientas a través de su API y, según sus creadores, permite desarrollar herramientas de modelado de forma rápida, intuitiva y a bajo coste. Para ello sigue un proceso de construcción en dos pasos:

1. Diseño del lenguaje de modelado (conceptos, reglas, notaciones gráficas y generadores) con *MetaEdit+ Workbench*. Al igual que Eclipse, *MetaEdit+* adopta la técnica del metamodelado, en base también a un lenguaje de metamodelado propio formalizado como metametamodelo y llamado GOPPRR (Grafo-Objeto-Puerto-Propiedad-Relación-Rol), según el cual la estructura de nivel superior de un metamodelo es un grafo. Así, la

[3] http://www.eclipse.org/modeling/
[4] http://www.eclipse.org/modeling/emf/
[5] http://metaphor.it.jyu.fi
[6] http://www.metacase.com/products.html

definición de un lenguaje se almacena en forma de metamodelo en el *MetaEdit+ Repository*. Una vez que el proyecto ha sido creado y registrado en el repositorio, puede utilizarse como editor.

2. Uso en *MetaEdit+ Modeler* del editor producido. Se sigue la definición del lenguaje de modelado previamente especificada en *MetaEdit+ Workbench*, extrayéndola del repositorio y se ofrecen funcionalidades de herramientas de modelado: editores de diagramas, visores, generadores, soporte multiusario, etc.

MetaEdit+ es un producto comercial que puede considerarse maduro y de uso amigable y está disponible para su utilización sobre las principales plataformas actuales. Hasta la aparición en 2006 de EMP y su posterior consolidación, *MetaEdit+* era la plataforma más sofisticada y conocida para desarrollar entornos DSM. En ella, tanto metamodelado como modelado tienen lugar en el mismo entorno integrado.

- ***Generic Modeling Environment* (GME, 2000 –).** Herramienta metaCASE configurable en la que, también empleando la técnica del metamodelado, la configuración se logra a través de metamodelos que especifican el paradigma de modelado (lenguaje de modelado) del dominio de aplicación. Su desarrollo (en C++) se debe al *Institute for Software Integrated Systems* (ISIS) de la Universidad de Vanderbilt [7] y, al igual que *MetaEdit+*, tiene su origen en un extenso historial de investigación previa y experiencia práctica en metamodelado y herramientas de modelado. El trabajo anterior llevado a cabo en Vanderbilt en el área de Arquitecturas MultiGrafo (MGA) [56] había servido para probar el valor del DSM y crear un *framework* para construir herramientas de modelado. Finalmente, en [57] se esbozó un metametamodelo apropiado para definir DSMLs dentro del ámbito de la ingeniería eléctrica así como un Entorno de Modelado Genérico que podía configurarse mediante metamodelos expresados con tal metalenguaje.

En el artículo seminal [58] se presenta GME y se compara con otras alternativas del momento (DoME), todo ello sobre la base de un caso de estudio que ilustra los conceptos básicos. Una visión general más reciente puede encontrarse en el informe técnico[8], en el que se introducen los conceptos y técnicas requeridos para desarrollar entornos DSM, argumentando que, para un amplio abanico de dominios, la utilización de un conjunto de herramientas configurable y extensible permite desarrollar rápidamente un entorno de modelado completamente funcional, logrando la configuración gracias a especificar lenguajes de modelado a través de metamodelos. Junto a este reporte, en el propio sitio web de GME se encuentra disponible una extensa guía de usuario.

Básicamente, el lenguaje proporcionado por GME para especificar metamodelos está basado en la notación de los diagramas de clases de UML y hace uso extensivo de estereotipos para distinguir los diversos metatipos así como de un dialecto de tipo OCL para especificar restricciones. Este lenguaje de metamodelado también viene formalizado como metametamodelo (denominado MetaGME) incorporado dentro del propio GME. Sin embargo, MetaGME difiere significativamente del metametamodelo de otras herramientas, pues no

[7] http://www.isis.vanderbilt.edu/projects/GME
[8] http://www.isis.vanderbilt.edu/sites/default/files/GMEReport.pdf

presenta concepto de contenedor de primer nivel (grafo, paquete, etc.) aunque es cierto que se permite a los objetos contener otros objetos para formar una jerarquía.

En el ámbito GME, un paradigma de modelado contiene toda la información sintáctica, semántica y visual relativa al dominio: qué conceptos serán utilizados para construir modelos, qué relaciones pueden existir entre tales conceptos, cómo pueden éstos ser organizados y visualizados por el modelador y las reglas que gobiernan la construcción de modelos. Así, un paradigma de modelado define la familia de modelos que pueden ser creados usando el entorno de modelado resultante. Los metamodelos que especifican el paradigma de modelado son convertidos a fichero binario usado para generar automáticamente el entorno específico de dominio (constituyendo una segunda instancia de GME), el cual a su vez se utiliza para construir modelos de dominio que se almacenan en una base de datos de modelos o en formato XML. Estos modelos son empleados para generar automáticamente las aplicaciones.

GME sólo está disponible para Windows y tiene una arquitectura modular que utiliza MS COM para integración, de forma que GME es fácilmente extensible, pudiéndose escribir componentes externos en cualquier lenguaje que soporte COM (C++, Visual Basic, C#, Python etc.).

- **Microsoft DSL Tools (2005 –)**. Infraestructura ofertada por Microsoft [9] para DSM en forma de una combinación de *frameworks*, lenguajes, editores, generadores y asistentes que permite especificar lenguajes de modelado y construir herramientas de soporte. *MS DSL Tools* forma parte de un proyecto más amplio llamado *Microsoft Software Factories* [19] y, naturalmente, sólo está disponible para *Windows*. De hecho, sus herramientas de modelado y metamodelado sólo funcionan como parte de *Visual Studio* (aparecieron por primera vez en el *Visual Studio 2005 SDK 3.0*).

Desde muy pronto, *Microsoft* argumentó contra la adecuación de MOF y UML para describir lenguajes de modelado. Sin embargo, paradójicamente el metametamodelo propuesto no es muy diferente de MOF y en realidad comparte muchos de sus defectos. Los metamodelos se especifican en una versión gráfica de este metametamodelo y son transformados a código C# y C++ que posteriormente es extendido con código manual relativo a restricciones, símbolos apropiados y mejora del comportamiento del editor. El resultado se compila, se construye y se abre como herramienta DSM en una segunda instancia (instancia de depurado) de *Visual Studio*.

- **Fujaba Tool Suite (2002 –)**. Conjunto de herramientas CASE de código abierto que proporciona soporte para el desarrollo basado en modelos. Su desarrollo y extensión se enmarca dentro del proyecto *Fujaba* [10], en el que actualmente participan diversas universidades de toda Alemania. Descendiente del entorno *FUJABA*, a raíz de su rediseño en 2002 se convirtió en *Fujaba Tool Suite*, con una arquitectura modular que lo convierte en una plataforma extensible a la que se puede añadir funcionalidad fácilmente [59].

De cara al desarrollo de sistemas *software*, *Fujaba Tool Suite* presenta como característica principal un potente lenguaje OO gráfico, sencillo a la vez que formal, para especificar completamente la estructura y conducta del sistema bajo desarrollo. Se combinan los diagramas de clases para especificación estructural con una especialización de los diagramas de actividad (*story diagrams*) para especificación de comportamiento. Los *story diagrams* son una

[9] http://www.microsoft.com/en-us/download/details.aspx?id=2379
[10] http://www.fujaba.de/

combinación de diagramas de actividad y reglas de transformación de grafos (*story patterns*). En base a esta especificación formal de la estructura y comportamiento de un sistema, *Fujaba Tool Suite* genera código Java.

La otra propiedad fundamental de *Fujaba Tool Suite*, y la interesante en el contexto de este trabajo, es su caracterización como *framework* extensible tras su rediseño a partir del entorno *FUJABA* original. Su arquitectura modular permite a los desarrolladores extenderla acorde a sus propios intereses y de hecho se encuentran disponibles numerosos *plugins* proporcionando soporte para ingeniería inversa, metamodelado MOF, transformaciones M2M, modelado y V&V de sistemas empotrados de tiempo real, etc.

En principio, el entorno *Fujaba Tool Suite* puro (plataforma básica) se ha distribuido como producto *standalone* pero desde 2006 existe una versión integrada en el IDE de Eclipse y llamada *Fujaba4Eclipse* (F4E). En el sitio web del proyecto *Fujaba* se avisa de que, habitualmente, las herramientas y resultados de investigación más recientes sólo están disponibles para FE4.

- ***MOFLON* (2002 -)**. Herramienta metaCASE que soporta el proceso MDSD proporcionando un *framework* para metamodelado y transformaciones de modelos. Desarrollado por el *Real-Time Systems Lab* de la Universidad de Darmstadt[11], su andadura comenzó en 2002, cuando la mayoría de las herramientas metaCASE existentes carecían de fundamentos formales y no eran conformes a estándares. El objetivo que se estableció era mejorar el soporte de herramientas para MDSD, proporcionando una herramienta metaCASE completa, con soporte para transformaciones de modelos formalmente fundamentadas y siendo sus principales premisas la conformidad con estándares, la reutilización de componentes externos estables y la integración con herramientas comerciales, así como altos grados de escalabilidad y usabilidad.

En la trayectoria de *MOFLON* pueden distinguirse dos etapas bien diferenciadas. En la primera, se escogió *Fujaba* como plataforma sobre la que construir *MOFLON*, valorando especialmente que proporcionaba un *framework* para IDE y un motor genérico de transformaciones de modelos basado en el formalismo de las gramáticas de grafos. Así, en 2004 se completó una versión beta de *MOFLON* como *plugin* de *Fujaba*, soportando generación de código JMI conforme a MOF 2.0 y serialización MOF-XMI. Una descripción exhaustiva de *MOFLON* v1.0 (2006) sobre la base de *Fujaba* puede encontrarse en [60], abarcando su arquitectura y la generación de herramientas de soporte al modelado a partir de la especificación de metamodelos, su enriquecimiento con restricciones, la definición de transformaciones de modelos mediante Gramáticas de Grafos Triples (TGG) [61], etc. Posteriores versiones estables hasta v1.5 (2010) fueron incorporando más funcionalidades, como por ejemplo chequeo de restricciones mediante Dresden OCL [62].

Tras la v1.5, *MOFLON* experimentó una remodelación, abandonando *Fujaba* y migrando a Eclipse, dando así lugar a *eMoflon*. En [63] puede encontrarse la relación de las principales razones que motivaron tal reforma y una descripción de la nueva arquitectura. A grandes rasgos, *eMoflon* consiste básicamente en un componente *front-end* (en forma de *add-in* de *Enterprise Architect*) para metamodelado y un componente IDE basado en Eclipse para trabajar con el código generado.

[11] http://www.moflon.org/

- **IBM Rational Software Architect (IBM RSA)**. Herramienta estrella de la línea IBM Rational[12] para diseño, modelado y desarrollo de *software*. El soporte ofrecido para modelado está basado en UML 2.x, incluyendo el mecanismo de extensión de UML basado en perfiles y se posibilita la generación de editores para tales perfiles. Por un lado, soportar tal mecanismo constituye la fortaleza de RSA, pues se beneficia de la generalidad, reputación y sintaxis concreta de UML. Sin embargo, al mismo tiempo, forzar a los usuarios a crear un perfil UML para sus dominios representa su debilidad, ya que UML contiene muchos conceptos no siempre apropiados a las necesidades particulares de un dominio específico.

IBM RSA es un entorno comercial, con buena y abundante documentación [64, 65] , construido sobre el IDE de Eclipse, heredando pues todas las posibilidades de personalización de Eclipse a través de su API.

- **Obeo Designer**. Se define como un diseñador gráfico para definir el modelo de dominio correspondiente a un vocabulario de negocio y para la creación a su vez de una herramienta gráfica específica capaz de manipular nativamente los conceptos del dominio problema. Proporciona un enfoque basado en el concepto de "punto de vista", referido a que un mismo elemento de modelo puede mostrarse mediante diferentes representaciones dependiendo del rol o actividad del usuario.

Obeo[13] ha desarrollado comercialmente esta herramienta apoyándose en el IDE de Eclipse, lo que, al igual que a IBM RSA, le permite heredar todas las posibilidades de personalización de Eclipse a través de su API. Es más, Obeo Designer se basa directamente en los componentes de EMP (EMF, GMF, EMF Compare, ATL, Acceleo, EEF), de forma que cualquier otro componente o herramienta basados en EMF puede ser fácilmente integrado en *Obeo Designer*.

Literatura de estudios comparativos

Un sondeo exhaustivo sobre plataformas de desarrollo de software que pueden ser empleadas para construir entornos de desarrollo específicos de dominio que soporten MDSE puede encontrarse en [66], abarcando GME, *MetaEdit+*, *MOFLON* y EMP, entre otras. La motivación del análisis comparativo realizado es elegir una herramienta para implementar el entorno COMDES [67]. Finalmente, la plataforma seleccionada es Eclipse, debido, entre otras razones, a que soporta una robusta definición de GUIs y un lenguaje flexible para definición de restricciones, y así mismo porque en ella se pueden construir e integrar fácilmente herramientas a través de su mecanismo de *plugins*. Es más, esto constituye una vía para integración de herramientas que permite la colaboración entre herramientas de desarrollo heterogéneas.

También en [68] se discute el estado del arte en cuanto a entornos de metamodelado, realizando una evaluación de su productividad y expresividad. Se cubren IBM RSA, GME, *MetaEdit+*, *Obeo Designer* y Eclipse GMP, si bien en este caso la orientación es hacia las posibilidades de tales plataformas para ser usadas en la construcción de editores gráficos específicos. Así, por ejemplo, la sección correspondiente a Eclipse EMP se centra en GMF y *Graphiti*.

[12] http://www-03.ibm.com/software/products/es/ratisoftarch
[13] http://www.obeo.fr/pages/products/en

De entre las plataformas consideradas anteriormente, las no construidas sobre Eclipse presentan como similitud con Eclipse/EMF la presencia de un metametamodelo base (MetaGME, GOPPRR, etc.). En [69] se analiza un conjunto de lenguajes de metamodelado (Ecore, MetaGME y GOPPRR, entre otros), se definen criterios de comparación y se comparan los metametamodelos seleccionados. El ánimo de tal estudio comparativo es proporcionar una base para resolver los problemas inherentes a la disponibilidad de muchas alternativas en cuanto a lenguajes de metamodelado, esto es, selección e interoperabilidad entre herramientas de metamodelado. Una comparación mucho más detallada pero restringida a Ecore y GOPPRR se presenta en [70] sobre la base de haber especificado, tanto usando Ecore como a través de GOPPRR, los conceptos PIM correspondientes a una herramienta MDSD propia destinada al diseño de sistemas de información: *IIS*Case*. También se presentan las diferencias principales entre ambos entornos de metamodelado (EMF y *MetaEdit+*).

También existen interesantes trabajos en cuanto al desarrollo de pasarelas de interoperatividad entre diversos lenguajes de metamodelado. Por ejemplo, en [71] Bézivin et al. desarrollan un puente entre GME y EMF mientras que en [72] plantean una solución de compatibilidad entre MS DSL Tools y EMF. En ambos casos trabajan sobre la base de la plataforma AMMA [73] propia de su grupo de investigación, y entre otras cosas, cuna de ATL. Por último, un estudio sobre el intercambio de metamodelos entre *MetaEdit+* y EMF se presenta en [74].

1.4.2 Eclipse Modeling Project

Actualmente no se discute la preponderancia de EMP como plataforma para desarrollo de soluciones DSM, lo cual se confirma al comprobar que algunas de sus alternativas como plataforma de metamodelado están basadas en el propio Eclipse. De hecho, en los anteriormente mencionados trabajos relativos al desarrollo de pasarelas de interoperatividad [71] [72] [73] [74] queda patente el interés por alcanzar la capacidad de establecer mecanismos de compatibilidad desde entornos no-EMF a EMF, a partir de lo cual puede reafirmarse la preponderancia de este último.

La principal diferencia entre EMP y sus competidores es que éstos son productos fruto del esfuerzo de una única institución o empresa y que, independientemente de si la capacidad de extensibilidad ha sido considerada un requisito importante durante su diseño y desarrollo, ofrecen de manera autocontenida toda la funcionalidad correspondiente a las áreas a las que se pretende dar cobertura. Fácilmente pueden considerarse superiores a EMF de forma aislada, pero el conjunto de aportaciones que conforman EMP enriqueciendo la funcionalidad base de EMF hacen que actualmente Eclipse no tenga rival como plataforma de metamodelado.

Relación con OMG

Una de las ventajas de EMF/Ecore es que está muy ligado a importantes estándares de modelado pertenecientes a OMG, incluyendo UML, MOF y XMI. De hecho, Ecore puede entenderse como un subconjunto pequeño y simplificado de UML, restringido básicamente al modelado de clases. Por tanto, más que a UML, Ecore es comparable a MOF, aunque, con un punto de mira más puesto en la integración de herramientas que en la gestión de repositorios de metadatos. Ecore evita algunas complejidades de MOF, resultando así en una implementación optimizada y ampliamente aplicable. Así, uno de los motivos por los que Ecore es tan utilizado es que, a falta de una implementación estandarizada de MOF, se puede

seleccionar Ecore como estándar *de facto* y a través suyo se tiene la oportunidad de modelar usando los conceptos de MOF. Es más, la experiencia en desarrollo con Ecore ha influenciado sustancialmente en la especificación MOF 2.0, en la cual se define por separado Essential MOF (EMOF), núcleo ligero del metamodelo y que recuerda fehacientemente a Ecore. Puesto que los dos son tan similares y poseen una relación y alineación tan estrecha, EMF soporta directamente EMOF como serialización para Ecore alternativa a XMI.

Kernel de Ecore

La Figura 1.3 muestra un subconjunto simplificado del metametamodelo Ecore (el llamado *kernel* de Ecore en [52]), que ilustra los principales conceptos de metamodelado incluidos.

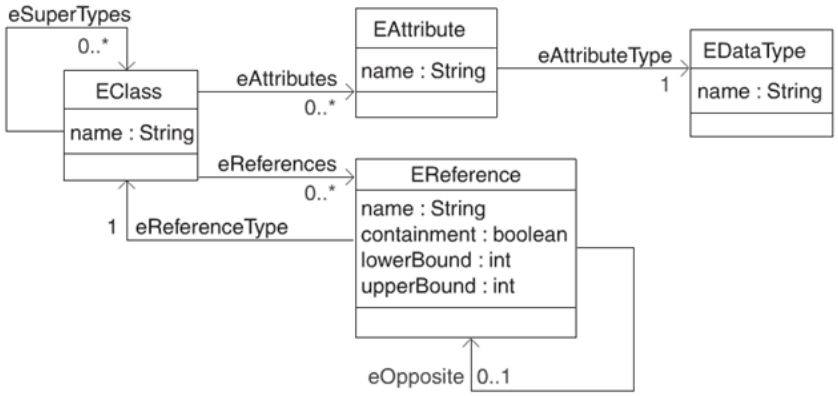


Figura 1.3 – Kernel de Ecore

- **EClass** modela los elementos de un metamodelo, esto es, las clases (instancias de EClass), las cuales se identifican por nombre y pueden tener diversos atributos (instancias de EAttribute) y referencias (instancias de EReference). Se observa que una clase puede referirse a un cierto número de supertipos, es decir, en Ecore existe herencia múltiple.
- **EAttribute** modela los atributos, representativos de los datos de un objeto. También se identifican por nombre y tienen tipo.
- **EDataType** se usa para representar tipos simples utilizados para tipar atributos. Al igual que las clases, también se identifican por nombre.
- **EReference** modela asociaciones entre clases, en concreto un extremo de una asociación y si la asociación es navegable en sentido opuesto, habrá otra referencia para representar esta bidireccionalidad. Al igual que los atributos, las referencias se identifican por nombre y tienen tipo, que ha de ser la clase en el otro extremo de la asociación. Una referencia especifica límites de multiplicidad y si es de tipo composición.

La descripción detallada del metametamodelo Ecore completo puede encontrarse en [52].

Infraestructura disponible alrededor de EMF / Ecore

Además de su simplicidad y afinidad con los esfuerzos de estandarización por parte de OMG, y en consecuencia familiaridad para desarrolladores con nociones de OO, el gran punto fuerte relacionado con el empleo de Ecore como lenguaje de metamodelado es el extenso conjunto de herramientas disponibles alrededor suyo. Es precisamente a través de los proyectos englobados por EMP como se organizan de forma lógica las distintas áreas cubiertas: desarrollo de sintaxis

abstracta, de sintaxis concreta (gráfica y textual), de transformaciones de modelos (tanto M2M como M2T), etc. La lista completa de proyectos aglutinados bajo EMP puede consultarse en http://projects.eclipse.org/projects/modeling.

El subproyecto nuclear EMF, a pesar de ser la cuna de Ecore, no se reduce a él. Sus creadores lo definen como "*un framework de modelado y generación de código para construir herramientas y aplicaciones basadas en un modelo de datos estructurado. A partir de una especificación de un metamodelo, EMF proporciona herramientas y soporte de runtime para producir un conjunto de clases Java para el metamodelo, junto a un conjunto de clases adaptadoras que habilitan su visualización y edición basada en comandos, así como un editor básico de modelos instancia*".

EMF permite la construcción directa, desde cero, de modelos Ecore (metamodelos), proporcionando para ello un editor simple de tipo árbol y más recientemente un editor gráfico; ambos operando en colaboración con la vista *Properties*. Además, también existe una sintaxis textual concreta denominada Emfatic [14] con soporte de editor textual.

Con el objetivo de ser atractivo a desarrolladores con experiencia en otros ámbitos y también de facilitar la reutilización de activos ya existentes, además de la creación directa, EMF contempla la obtención de modelos Ecore mediante un proceso de importación sobre otra representación del mismo modelo conceptual. Así, los metamodelos también pueden ser especificados mediante código Java, W3C-Schema o diagramas de clases UML y posteriormente ser importados a Eclipse. Por ejemplo, si se está más familiarizado con la codificación en Java que con herramientas de modelado, usar código Java podría resultar la manera más fácil de describir un metamodelo. De cara a EMF, esto es posible mediante interfaces Java especialmente anotadas, en cuyo caso el correspondiente asistente de importación en EMF las inspecciona y construye el modelo Ecore. Si se trabaja con modelos expresados mediante documentos XML relativos a un W3C-Schema, entonces EMF es capaz de construir el modelo Ecore semánticamente equivalente a partir de tal *schema* y sus modelos-instancia serán conformes al *schema* cuando se serialicen a XML. En caso de que el escenario inicial de trabajo sea UML, existen dos opciones para migrar a EMF los modelos existentes. Por un lado, el asistente de importación proporcionado por EMF permite importar desde Rational Rose (ficheros *.mdl) y por otro lado, existen herramientas UML que permiten exportar a Ecore.

Puesto que Ecore tiene sus raíces en UML y MOF, está diseñado para mapear limpiamente a implementaciones en lenguajes OO como Java. Así, a partir de un modelo Ecore, EMF es capaz de generar para él una implementación en Java que proporciona un API a medida para construir programáticamente modelos-instancia. Además, Ecore soporta conceptos que no están directamente incluidos en Java (como la herencia múltiple), siendo esta habilidad para generar implementaciones Java correctas y eficientes de tales construcciones gran parte del valor de EMF. A esto se añade que también está equipado con dos generadores adicionales que producen respectivamente un *framework* genérico para editado de modelos y editores. Utilizando este API a medida automáticamente generado resulta sencillo el *parseo* de modelos específicos conformes a modelos Ecore. Lo realmente importante es que EMF proporciona la base para la interoperabilidad entre herramientas y aplicaciones basadas en él.

[14] http://projects.eclipse.org/projects/modeling.emft.emfatic

Además del núcleo original, existen varios componentes disponibles por separado dentro de EMF, los cuales extienden y complementan sus capacidades básicas, como, por ejemplo, los componentes *Model Transaction*, *Model Validation* y *Model Query*. El primero proporciona soporte transaccional a la hora de editar modelos Ecore, mientras que el segundo es un *framework* que complementa al anterior, proporcionando soporte de integridad. En cuanto a *Model Query*, puesto que, al igual que en una base de datos, es comúnmente necesario consultar los contenidos de un modelo, este componente proporciona para ello alternativas de tipo OCL y SQL, frente a las API Java proporcionadas por defecto en EMF. La lista completa de componentes de EMF puede consultarse en http://projects.eclipse.org/projects/modeling.emf

Sintaxis concreta

Además de la serialización XMI proporcionada por defecto por EMF – tanto de metamodelos como de modelos-instancia –, los proyectos *Graphical Modeling Project* (GMP) y *Textual Modeling Framework* (TMF) dentro de EMP proporcionan la posibilidad de desarrollar sintaxis concretas de tipo gráfico y textual, respectivamente.

GMP posibilita un rápido desarrollo de editores gráficos de modelado según el estándar Eclipse. Para ello, en primer lugar permite crear una notación gráfica para un lenguaje de modelado, mapearla a su sintaxis abstracta y por último generar un rico editor gráfico para el lenguaje en cuestión. Este proyecto está constituido por subproyectos concretos *Graphical Modeling Framework* (GMF Notation, GMF Runtime y GMF Tooling) y Graphiti. Paralelamente, TMF aloja *frameworks* de modelado textual con los que producir editores textuales. Contiene dos proyectos concretos: *Xtext* y *Textual Concrete Syntax* (TCS) – actualmente inactivo –.

Transformaciones de modelos

En este ámbito, EMP proporciona soluciones tanto M2M como M2T. El proyecto *Model-to-Model Transformation* (MMT) – anteriormente llamado M2M – alberga componentes de lenguajes de transformación M2M. Naturalmente, QVT, como estándar de OMG para transformaciones de modelos, y también la alternativa ATL. Además, otros proyectos EMP relacionados con M2M y con entidad propia (no contenidos en MMT) son Epsilon, Viatra2 y Xtend2.

Paralelamente, el proyecto M2T se centra en la generación de artefactos textuales a partir de modelos, englobando 3 componentes: *Java Emitter Templates* (JET) – quizás el mejor conocido, usado por el propio EMF –, *Acceleo* y *XPand* (motor de plantillas usado extensivamente por GMF).

Otros proyectos relevantes

Otros proyectos interesantes dentro de EMP y que no pertenecen a ninguna de las categorías vistas son los proyectos *Model Development Tools* (MDT) y *EMF Technology*.

El proyecto MDT está orientado a dar soporte a metamodelos estándares, como los producidos por OMG. Tiene el doble propósito de proporcionar tanto una implementación de tales metamodelos estándares como herramientas para desarrollar modelos basados en ellos. MDT consta de varios componentes, como por ejemplo, UML2 y EclipseOCL. UML2 proporciona una implementación basada en EMF del metamodelo UML 2.x de OMG, y, al haber sido desarrollado en colaboración con la propia especificación, es *de facto* su implementación de referencia.

EclipseOCL proporciona una implementación de la especificación OCL 2.0 de OMG así como *bindings* para Ecore y UML2. El núcleo de este componente OCL proporciona capacidades para soportar la integración de OCL, como por ejemplo APIs para *parsear* y evaluar restricciones y consultas OCL en modelos Ecore o UML o implementaciones Ecore y UML de la sintaxis abstracta de OCL, incluyendo soporte para la serialización de expresiones OCL *parseadas*.

El proyecto *EMF Technology* – anteriormente llamado *Generative Modeling Technologies* (GMT) – está destinado a incubar nuevas tecnologías que extiendan o complementen a EMF. Tales componentes son posteriormente liberados y se convierten en subproyectos de EMP de pleno derecho o de algún otro proyecto dentro de EMP.

1.4.3 Entornos orientados a sistemas distribuidos de tiempo real

En esta sección se exponen algunos entornos de desarrollo de STRE fundamentados en la metodología MDSE y comúnmente basados en un metamodelo que conceptualiza el dominio específico de comportamiento temporal. En algunos casos, su soporte es alguna de las plataformas de desarrollo presentadas en la subsección 1.4.1:

- CoSMIC
- OSATE
- TOPCASED
- TASTE
- Fujaba Real-Time Tool Suite
- SCADE
- STOOD
- BridgePoint

A continuación se presentan brevemente:

- **CoSMIC (*Component Synthesis using Model Integrated Computing*)** [75]. *Suite* de herramientas MDD enfocada a los principales retos del desarrollo basado en componentes de sistemas DREs, permitiendo especificar, desarrollar, componer e integrar *software* tanto a nivel de aplicación como de *middleware*. CoSMIC[15] es un entorno de código abierto muy vinculado a los estándares de OMG y desarrollado por la Universidad de Vanderbilt sobre su propia plataforma GME (v10.8.18), es decir, sus herramientas MDD están implementadas mediante paradigmas de modelado desarrollados usando GME. Una propiedad interesante es que posee la capacidad de interactuar con herramientas de chequeo de modelos, como Cadena [76].

- **OSATE (Open Source AADL Tool Environment)**. El *Software Engineering Institute* (SEI) de la Universidad Carnegie Mellon[16] ha desarrollado OSATE, un completo entorno para modelado mediante AADL [77], cubriendo tanto el núcleo del lenguaje como varios de sus anexos (sublenguajes), como por ejemplo el de errores o el de comportamiento. OSATE es un producto de código abierto que ha sido construido sobre Eclipse y que se encuentra disponible para las principales plataformas (Windows, Mac OS y Linux). Mediante su uso, los desarrolladores arquitecturales pueden escribir sus modelos empleando la sintaxis textual de AADL y también visualizarlos mediante una notación gráfica.

[15] http://www.dre.vanderbilt.edu/cosmic/
[16] http://www.sei.cmu.edu/architecture/tools/analyze/index.cfm

OSATE ofrece muchas facilidades para modelar y analizar sistemas, y además sus puntos de extensión permiten a los usuarios extender su capacidad inicial construyendo sus propias herramientas AADL sobre él, sin tener que diseñar todo desde cero. De hecho, existen diversos *plugins* para OSATE liberados bajo la licencia pública de Eclipse (EPL). Pueden citarse RDALTE (*Requirements Definition and Analysis Language Tool Environment*), para conectar modelos AADL con documentos de requisitos, IMV (*Instance Model Viewer*), que proporciona un visor que posibilita una visualización gráfica de modelos AADL, Lute, un lenguaje de restricciones que permite verificar las características de un modelo según los requisitos de diseño y/o contratos y RC Meta.

Bajo la premisa inicial de brindar soporte a AADL en Eclipse, OSATE también facilita la conexión con herramientas existentes. Por ejemplo, OSATE está integrado con la herramienta europea de código abierto TOPCASED (ver a continuación).

- **TOPCASED (*The open source toolkit for critical systems*)** [78]**.** Entorno de herramientas de desarrollo orientado a sistemas críticos y empotrados (*software* y *hardware*) que cubre diversas fases de su ciclo de desarrollo, desde análisis de requisitos a implementación, así como otras actividades transversales como gestión de anomalías, control de versiones y trazabilidad de requisitos. TOPCASED está fuertemente orientado a modelado, no proporcionando únicamente editores de modelos y transformaciones, sino que el propio TOPCASED está también basado en modelado y generación de código. Actualmente en proceso de migración a PolarSys[17], TOPCASED está construido sobre Eclipse, es un producto de código abierto e incluye OSATE.

- **TASTE** [79-81] Conjunto de herramientas libres dedicadas al desarrollo basado en AADL de SDTREs. Fue desarrollado por la Agencia Espacial Europea (ESA), en colaboración con socios de la industria aeroespacial. TASTE permite a los diseñadores de software integrar fácilmente herramientas de modelado externas, como MATLAB, Simulink, SCADE o Real-Time Developer Studio.

- ***Fujaba Real-Time Tool Suite***. En el entorno *Fujaba Tool Suite* existen distribuciones especializadas construidas sobre la plataforma *Fujaba Tool Suite* básica, como por ejemplo *Fujaba Real-Time Tool Suite* [82], que añade herramientas de modelado, verificación y generación de código para sistemas empotrados de tiempo real. Esta *suite* se basa en *MechatronicUML*[18] (MUML) [83], un DSL gráfico destinado al desarrollo de *software* para sistemas mecatrónicos[19] avanzados, que trata explícitamente el modelado de restricciones de tiempo real y adaptación en tiempo de ejecución. *Fujaba Real-Time Tool Suite* supone la implementación de referencia de MUML, proporcionando una formulación EMF de su metamodelo y editores gráficos para crear modelos MUML. Al igual que la plataforma base, en principio esta *suite* está disponible como *standalone* pero a raíz de la liberación de Fujaba4Eclipse también está implementada como un conjunto de *plug-ins* Eclipse.

[17] http://www.topcased.org/index.php/content/view/40/54/

[18] https://trac.cs.upb.de/mechatronicuml/

[19] En este contexto se define sistema mecatrónico como sistema que combina partes mecánicas, eléctricas, de ingeniería de control y de *software* que interaccionan constantemente. Son sistemas expuestos a restricciones de tiempo real estricto, las cuales incrementan la complejidad del desarrollo de *software*. Además, los sistemas mecatrónicos avanzados poseen la habilidad de adaptarse en tiempo de ejecución a un entorno variable.

- **SCADE**. La familia de productos SCADE, comercializada por *Esterel Technologies*[20], es una solución formal, exhaustiva y probada a nivel industrial para desarrollo de sistemas *software* críticos, soportando todo el ciclo de desarrollo, desde análisis de requisitos hasta diseño, verificación, implementación y despliegue. Dentro de esta línea de productos, SCADE System es una *suite* de herramientas de modelado y diseño de sistemas que ha sido desarrollada específicamente para ser utilizada en sistemas críticos con requisitos *high dependability*, proporcionando soporte completo a procesos industriales de ingeniería de sistemas.

- **STOOD**. *Suite* de herramientas para *software* de tiempo real comercializada por Ellidiss[21] y que soporta un amplio rango de métodos y lenguajes, incluyendo AADL, HOOD, UML, C, C++ y Ada. STOOD permite integración con OSATE y TOPCASED y es comúnmente utilizado en el desarrollo de sistemas espaciales y de aviación, por ejemplo ha sido la principal herramienta de diseño usada por AIRBUS.

- ***BridgePoint*.** *Suite* de herramientas UML debida a *Mentor Graphics*[22] y orientada a ingenieros de sistemas, *software* y *hardware* que desarrollen sofisticados sistemas embebidos. *BridgePoint* ofrece un entorno de diseño *model-driven* utilizando la potencia de xtUML [84].

1.5 El Entorno MAST como oportunidad de aplicación de MDSE

1.5.1 Visión general del Entorno MAST

MAST[23] (*Modeling and Analysis Suite for Real-Time Applications*) es un entorno para diseño y análisis de SDTREs, desarrollado por el Grupo ISTR[24] de la Universidad de Cantabria. La Figura 1.4 muestra la composición del Entorno MAST. Como puede observarse, está básicamente constituido por una terna de modelos conceptuales (metamodelos) y un conjunto de herramientas que orbitan a su alrededor.

- **Metamodelos**. Constituyen el núcleo del Entorno MAST, pues representan la descripción de los datos gestionados por sus herramientas.

- **Herramientas**. Operan sobre datos formalizados según tales metamodelos y se clasifican en tres categorías principales:

  - De análisis de planificabilidad y diseño de planificación.
  - De gestión de datos.
  - De soporte a paradigmas avanzados de diseño TR.

  Las categorías más relevantes en el contexto de esta Tesis son la de gestión de datos y la de soporte a paradigmas avanzados de TR, junto a las herramientas de simulación, las cuales, como se expondrá a continuación, forman parte de la categoría de herramientas de análisis de planificabilidad. La adopción de MDSE supone inmediatamente consecuencias altamente beneficiosas en todas ellas, como se verá en la subsección 1.5.3.

[20] http://www.esterel-technologies.com/products/
[21] http://www.ellidiss.com/products/
[22] http://www.mentor.com/products/sm/model_development/bridgepoint/
[23] http://mast.unican.es
[24] http://www.istr.unican.es

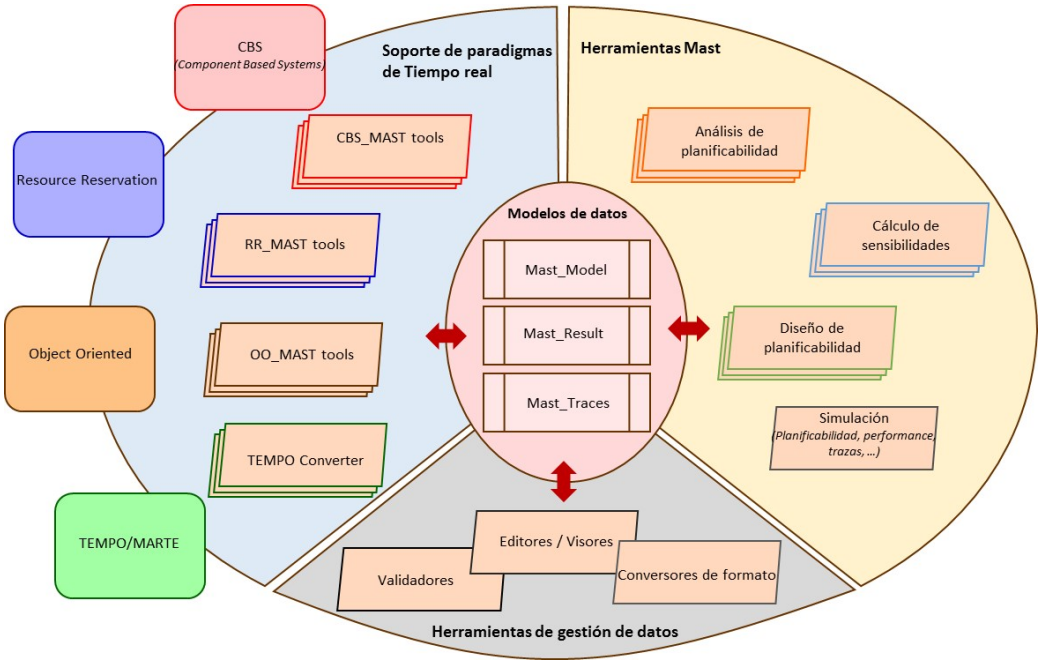


Figura 1.4 – Visión general del Entorno MAST

Modelos conceptuales

Dentro de la terna de metamodelos, el principal es el metamodelo MAST.

- **Metamodelo MAST**. Está orientado a la especificación del comportamiento temporal de SDTREs mediante un enfoque reactivo. Se trata de un modelo conceptual muy próximo al definido por el componente SAM dentro del perfil MARTE [5].

  La exposición detallada de los elementos de modelado incluidos puede encontrarse en el documento *Description of the MAST Model* [85].

Los modelos de comportamiento temporal conformes al metamodelo MAST son procesados por las herramientas de análisis del entorno, las cuales generan unos datos de salida: *los resultados del análisis*. Además, las herramientas de análisis mediante simulación generan otro conjunto adicional de datos: *las trazas de simulación*. Para dar soporte a tales conjuntos de datos son necesarios los otros dos modelos conceptuales, que complementan al metamodelo MAST.

- **Metamodelo de Resultados**. Modelo conceptual relativo a la descripción de los resultados de análisis de planificabilidad.
- **Metamodelo de Trazas**. Modelo conceptual relativo a la descripción de escenarios de comportamiento de los sistemas, formulados como secuencias de cambios de estado que se han producido en el sistema durante su ejecución y a los que se les ha asociado el tiempo en el que se ha producido.

Herramientas de análisis y configuración

El núcleo de herramientas que justifican la existencia del entorno MAST son las dedicadas al análisis de planificabilidad y configuración de planificación. Éstas pueden a su vez dividirse en las siguientes subcategorías:

- **Análisis de planificabilidad (técnicas analíticas)**. Estas herramientas llevan a cabo diferentes tipos de análisis de peor caso para determinar la planificabilidad del sistema.

  La descripción detallada de las diversas técnicas analíticas soportadas puede encontrarse en el documento *Analysis Techniques used in MAST* [86].

- **Simulación**[25] [26]. Estas herramientas son capaces de simular la conducta del sistema para chequear requisitos temporales laxos y generar trazas temporales.

- **Análisis de sensibilidad**. Estas herramientas calculan las holguras (*slacks*) correspondientes a recursos de procesamiento, a transacciones o al sistema global, informando así de qué partes del sistema tienen más espacio para crecer o cuáles necesitan ser modificadas para que el sistema sea planificable.

- **Asignación de parámetros de planificación**. Estas herramientas son capaces de hacer una asignación automática de diversos parámetros de planificación, como pueden ser prioridades, techos de prioridad, *deadlines*, etc.

Herramientas de gestión de datos

Esta categoría engloba las herramientas dedicadas al manejo de la información con que operan las herramientas de análisis, formulada como modelos. Básicamente se trata de editores y visores de modelos así como varios conversores de formato.

Herramientas de soporte a paradigmas de diseño TR

MAST incluye un conjunto de herramientas destinadas a facilitar la aplicación de sus recursos de análisis y diseño de planificabilidad a sistemas desarrollados siguiendo diversos paradigmas de diseño de software, bien propios del ámbito TR (Reserva de Recursos) o bien genéricos a los que se desea incorporar TR (OO, componentes, etc.). Entre ellos se han abordado:

- **CBS-MAST**. Herramienta cuyo objetivo es incorporar las tareas de análisis y diseño TR a los procesos de *Desarrollo de Aplicaciones Basado en Componentes* (CBSE [87]). Para ello, la herramienta implementa la estrategia descrita en [88], la cual posibilita que los diseñadores no tengan que construir el modelo de comportamiento temporal (o modelo TR) de la aplicación (ensamblado de componentes) ,anualmente, sino que el aspecto clave de la estrategia diseñada es la generación automática del modelo TR de la aplicación mediante composición de los modelos TR de los componentes y de los recursos de la plataforma, modelos accesibles desde las descripciones del plan de despliegue, de la carga de trabajo y de la propia plataforma de ejecución.

- **RR-MAST**. Conjunto de herramientas de soporte al diseño de SDTREs estricto que se ejecutan en plataformas distribuidas y abiertas, en base al paradigma de Reserva de Recursos (RR [89]). Para ello, las herramientas implementan la estrategia descrita en [90].

- **RTOO-MAST**. Herramienta de soporte al diseño de aplicaciones OO de TR laxo que se ejecutan en sistemas embebidos, en base a RT-Java. Para ello, la herramienta implementa la estrategia descrita en [91], la cual propone un *framework* que formaliza el uso de los

[25] http://mast.unican.es/jsimmast/index.html
[26] http://mast.unican.es/simmast

patrones TR incluidos en RTSJ[27]. El objetivo es sistematizar el proceso de modelado y configuración de planificación de tales aplicaciones, posibilitando la generación automática del correspondiente modelo de comportamiento temporal.

- **TEMPO-MAST**. Ejemplo de conversor que permite aplicar las capacidades de análisis y diseño de planificabilidad del Entorno MAST a sistemas cuyo comportamiento temporal está modelado en base a una implementación que sintetiza el modelo de análisis de planificabilidad del perfil MARTE. En este caso se trata de la implementación TEMPO-MARTE propiedad de Thales[28] (Thales Research & Technology, TRT).
- Existen otras herramientas, como **UML-MAST**, **MARTE-MAST**, etc. que aún no han sido abordadas desde un punto de vista MDSE.

1.5.2 Evolución del Entorno MAST

La primera versión del Entorno MAST data del año 2000 y a partir de ahí ha ido evolucionando de manera uniforme hasta que actualmente se encuentra disponible la versión 1.5.1.0. La cobertura del metamodelo MAST se ha mantenido prácticamente estable desde el inicio, mientras que la evolución del entorno ha consistido en el paulatino enriquecimiento de la capacidad de las herramientas de análisis para cubrir todos los elementos de modelado.

En las sucesivas versiones MAST-1.x, los metamodelos nucleares se encuentran formulados mediante UML y también en forma de *W3C-Schema* (*Mast_Model.xsd* [29], *Mast_Result.xsd* [30] y *Mast_Trace.xsd* [31]). En cuanto a la formulación de los modelos, tanto de comportamiento temporal como de resultados y de trazas, dos han sido las estrategias empleadas:

- **Dialecto XML**. Sintaxis formalizada mediante los correspondientes W3C-Schema antes mencionados.
- **Lenguaje textual específico**. Sintaxis formalizada mediante plantillas textuales.

Recientemente se ha procedido a una revisión mayor del metamodelo MAST, dando como resultado la versión MAST-2 [92]. Esta última incluye todos los elementos disponibles en MAST-1.x (con algunos nombres modificados, buscando mayor alineación con el perfil MARTE) y lo extiende con nuevas primitivas de modelado, con el propósito de cubrir aspectos y paradigmas avanzados de diseño TR. Por ejemplo, MAST-2 cubre el paradigma de Reserva de Recursos, el estándar AFDX (*Avionics Full-Duplex Switched Ethernet*) o la planificación particionada en el tiempo.

Paralelamente, el Grupo ISTR ha participado durante los últimos años en los proyectos RT-Model (2011) y M2C2 (2015), desarrollando métodos y técnicas de diseño de sistemas ciberfísicos sobre plataforma multinúcleo que soporten la ejecución de SDTREs con diferentes niveles de criticidad. El Entorno MAST ha representado en ambos proyectos un escenario muy adecuado para adoptar MDSE como base de los productos desarrollados y poder así aprovechar las ventajas y beneficios que esta disciplina conlleva. En concreto, la evolución a MAST-2 ha

[27] http://www.rtsj.org/
[28] https://www.thalesgroup.com
[29] http://mast.unican.es/xmlmast/xmlmast_1_4/Mast_Model.xsd
[30] http://mast.unican.es/xmlmast/xmlmast_1_4/Mast_Result.xsd
[31] http://mast.unican.es/xmlmast/xmlmast_1_4/Mast_Trace.xsd

supuesto un importante revulsivo para rediseñar el entorno aplicando MDSE. En consecuencia, la nueva versión ha sido concebida para que opere sobre la plataforma Eclipse/EMF.

Son numerosos los aspectos del Entorno MAST que constituyen una oportunidad de introducir como base la disciplina MDSE y hacer uso de sus métodos y estrategias en la implementación de los servicios y funcionalidades que las herramientas del entorno proporcionan. En la siguiente subsección se presenta una visión de alto nivel de algunos de tales aspectos, donde las ventajas de su aplicación resultan más evidentes.

1.5.3 Aspectos del Entorno MAST donde aplicar técnicas MDSE

En principio, las secciones del Entorno MAST donde se ha detectado una mayor necesidad de la disciplina MDSE han sido las herramientas dedicadas a la gestión de datos y las dedicadas a dar soporte a paradigmas avanzados de diseño TR, junto a las de simulación. La reconversión en sentido MDSE estricto del bloque de herramientas de análisis aún se encuentra en fase de incubación, por lo que tales herramientas aún operan sobre la versión 1.x del metamodelo MAST, existiendo herramientas de conversión al efecto.

Desplegar una estrategia global MDSE sobre la que asentar MAST-2 requiere en primer lugar reformular los modelos conceptuales (núcleo del entorno) en términos del *framework* MDSE concreto elegido para construir la nueva versión del entorno, en este caso la plataforma Eclipse/EMF. Por tanto, los metamodelos del núcleo de MAST-2 han sido formulados como modelos Ecore. A ellos se dedica el siguiente apartado.

Formulación Ecore de los modelos conceptuales

El núcleo del Entorno MAST-2 es análogo al de su predecesor, si bien en este caso, al estar construido sobre la base de EMF, se toma como formulación principal de los metamodelos la basada en Ecore. Para cada uno de ellos se encuentra accesible su serialización Ecore-XMI.

- **Mast2.ecore**[32].

  El metamodelo MAST-2 se ha especificado de forma laxa, lo cual tiene pros y contras. En particular, tiene el inconveniente de que modelos conformes a él exhiban incoherencias. Sin embargo, trabajando en un marco MDSE, es posible la especificación formal de restricciones (por ejemplo, mediante OCL) tanto intrínsecas como específicas de herramienta o de paradigma de diseño. Este tema se aborda en profundidad en la sección 2.3.

- **Mast2_Result.ecore**[33].

  El metamodelo de resultados ha sido ampliado a los aspectos incluidos en MAST-2 y ha sido extendido para que de forma flexible pueda contener nuevos datos que sean generados por futuras herramientas.

- **Mast2_Trace.ecore**[34].

  El metamodelo de trazas ha sido modificado para que represente trazas con gran volumen de datos de forma más compacta. La información de trazas se descompone en un modelo con la información declarativa de cabecera que permite interpretar una instancia de las

[32] http://mast.unican.es/ecoremast/Mast2.ecore
[33] http://mast.unican.es/ecoremast/Mast2_Result.ecore
[34] http://mast.unican.es/ecoremast/Mast2_Trace.ecore

trazas y múltiples ficheros con segmentos incrementales de trazas formulados de forma compacta.

Gestión de datos

Las herramientas de gestión de datos constituyen el bloque del Entorno MAST en que está más generalizada la aplicación de la disciplina MDSE. De hecho, la concepción de la nueva versión MAST-2 en base a una infraestructura MDSE como Eclipse/EMF permite superar carencias y dificultades propias del ámbito MAST-1.x relativas a la gestión de los modelos de comportamiento temporal (en especial su construcción).

Por ejemplo, en MAST-1.x, en caso de decantarse por el lenguaje de propósito especial para la especificación de modelos y afrontar la edición textualmente[35], la especificación informal de la sintaxis del lenguaje mediante plantillas ha impedido contar con un editor textual específico que proporcione asistencia para la introducción de datos y permita la validación sintáctica en vivo. Además, tampoco hay herramientas que posibiliten las verificaciones sintácticas, de coherencia intrínseca, de consistencia respecto del paradigma de diseño elegido ni respecto de la herramienta que va a procesar el modelo, por lo que estas verificaciones tienen que ser incorporadas a cada herramienta que se desarrolle.

Al basar el Entorno MAST-2 en Eclipse/EMF se han obtenido evidentes beneficios en cuanto a usabilidad y productividad, a varios niveles:

- **Facilidades de gestión de modelos ofrecidas por defecto por la plataforma**. Se trata de herramientas genéricas (basadas en el metametamodelo Ecore) aplicables a modelos independientemente de su metamodelo.

  Ejemplos de tales herramientas con que viene equipado Eclipse/EMF son el editor reflexivo de tipo árbol de agregación (*Sample Reflective Ecore Model Editor*) o las funcionalidades de validación estructural de modelos respecto a su metamodelo y de comparación y fusión de modelos.

- **Desarrollo semiautomático de herramientas específicas que se integran perfectamente en la plataforma**. Dado un metamodelo, EMF y sus componentes satélites ofrecen distintas posibilidades de creación semiautomática de herramientas para gestión de los modelos conformes a él. Una vez generada la estructura básica de la herramienta, es posible su personalización y enriquecimiento según las propias necesidades.

  Ejemplos de tales situaciones son la posibilidad de desarrollar un editor de tipo árbol o un editor textual específicos para un metamodelo determinado. En el primer caso, gracias al propio EMF y en el segundo gracias al componente Xtext (ver subsección 2.4.4), es posible la obtención automática de tales editores a partir del metamodelo, totalmente funcionales. Posteriormente, es relativamente sencilla su personalización y enriquecimiento, por ejemplo, para que el editor sea inteligente y no ofrezca la posibilidad de construir modelos incorrectos respecto a un conjunto de restricciones. Ambos tipos de editores especializados han sido desarrollados para MAST-2 y se muestran en la Figura 1.5.

[35] Enfoque más conveniente que el gráfico en caso de modelos a partir de un cierto grado de complejidad y extensión. En la sección 2.4 se profundiza en las ventajas del modelado textual.

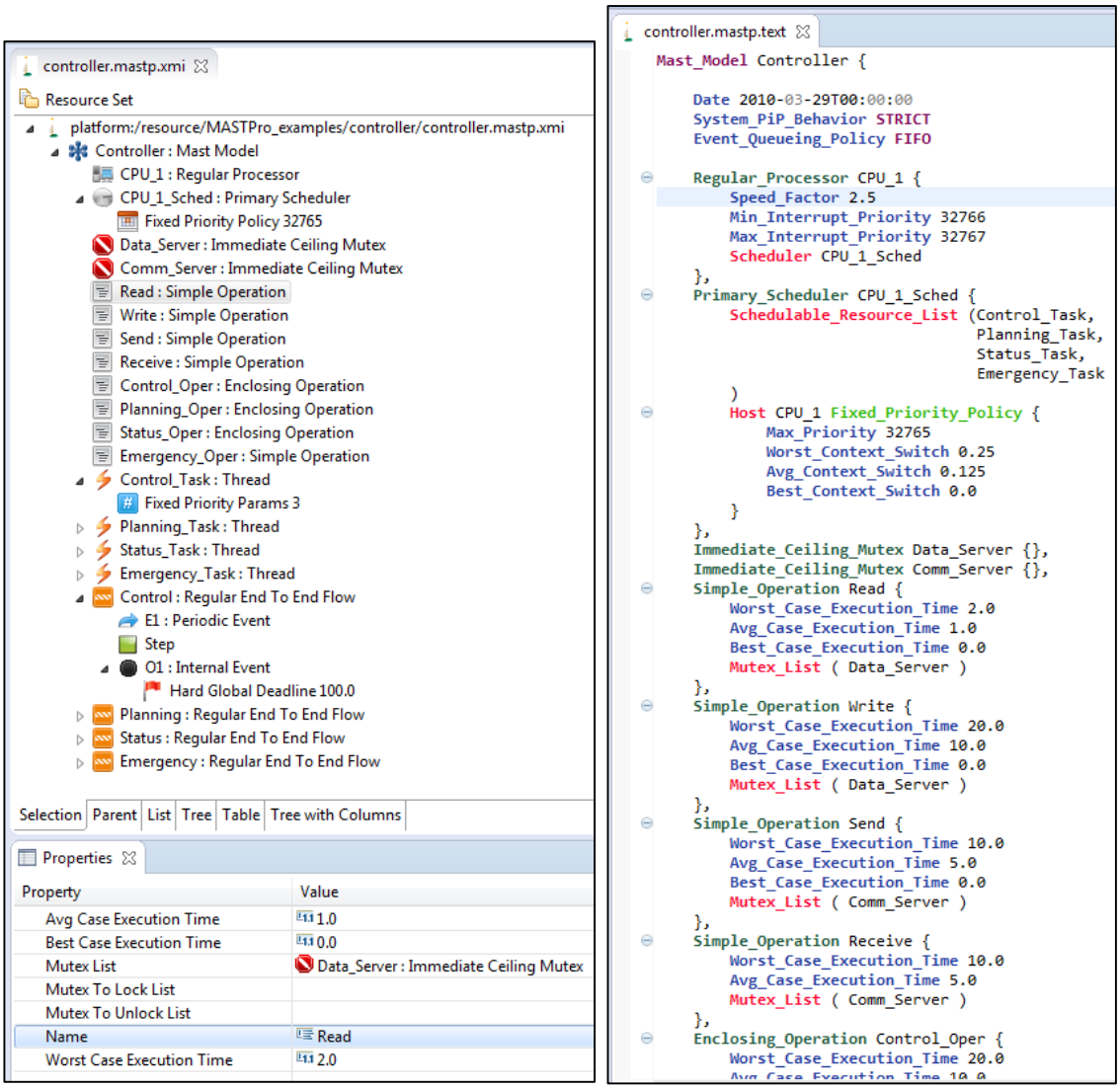


Figura 1.5 – Editores en árbol y textual para modelos MAST-2

- **Implementación de estrategias sofisticadas de manejo de la información**. Ejemplos de estrategias sofisticadas son la construcción de modelos en base a vistas especializadas del metamodelo o la obtención del resultado de verificar el cumplimiento de restricciones por parte de un modelo en forma de otro modelo para su posterior explotación o integración en procesos MDSE. Ambas han sido detectadas como necesidades relativas a la gestión de modelos en el Entorno MAST y han dado origen al desarrollo de sendas estrategias y metaherramientas, las cuales se exponen en el capítulo 3.

Soporte a paradigmas de diseño TR

Las herramientas del Entorno MAST destinadas a dar soporte a paradigmas avanzados de diseño TR introducidas en el correspondiente apartado de la subsección 1.5.1 implementan estrategias en su mayor parte concebidas (o al menos susceptibles de ser concebidas o reinterpretadas) desde una perspectiva MDD. Así, tales estrategias son fundamentalmente procesos *model-driven* que se articulan a través de elementos y mecanismos típicos en MDSE, como pueden ser los modelos, las transformaciones de modelos, la trazabilidad entre modelos, etc.

Bajo esta premisa, la opción más recomendable para implementar dichas herramientas es la aplicación de técnicas MDSE rigurosas. Por tanto, la implementación se simplifica si el Entorno MAST opera sobre la base de Eclipse/EMF y hace uso de sus componentes de modelado asociados, como, por ejemplo, lenguajes específicos de transformación de modelos. Además, las necesidades de trazabilidad entre modelos se ven soportadas por la formulación de referencias mediante URIs, un robusto mecanismo del que EMF hace uso extensivo.

Todo esto motiva que el Entorno MAST resulte muy apropiado como candidato a adoptar MDSE como metodología base y ser banco de pruebas de las diversas tecnologías MDSE evaluadas en el contexto del proyecto RT-Model.

Como ejemplo, la Figura 1.6 muestra el proceso *model-driven* soportado por la herramienta XXXXX-MAST para la integración de las técnicas de análisis de planificabilidad de MAST al entorno XXXXX de una empresa que va a hacer uso del entorno MAST.

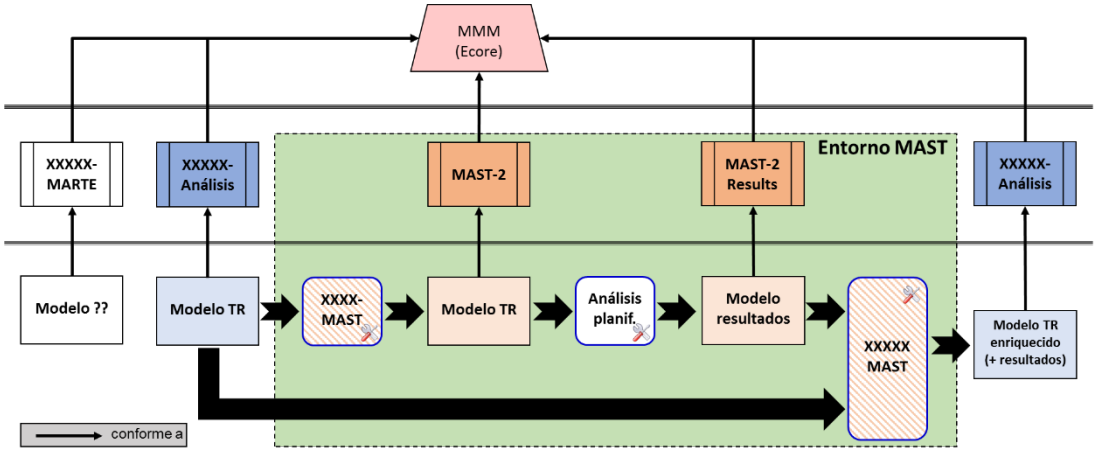


Figura 1.6 – Herramienta XXXXX-MAST

La herramienta se basa en la formulación Ecore de los metamodelos involucrados y en la codificación de las diversas transformaciones M2M mediante un lenguaje específico, como puede ser ATL.

1.6 Objetivos

A lo largo de este capítulo introductorio se han definido los conceptos que sirven de punto de partida al trabajo desarrollado en esta Tesis. En esta sección se exponen los objetivos establecidos en la fase de planteamiento y que han conducido su desarrollo. Sin embargo, antes de ser expuestos y para contextualizarlos adecuadamente, se introduce la siguiente subsección, dedicada a los roles involucrados en el ámbito de los entornos.

1.6.1 Agentes relacionados con los entornos de desarrollo

En el ámbito de los entornos de desarrollo pueden identificarse, en principio, dos roles básicos, mostrados en la Figura 1.7.

- **Desarrollador de sistemas software**. Usuario final de un entorno que lo emplea para desarrollar software en cualquiera de los aspectos o dominios soportados por éste (ingeniería de requisitos, de sistemas, diseño arquitectural, de componentes, de integración, etc.). En esencia, el propósito de un entorno no es otro sino proporcionar recursos a ser utilizados por este agente para que contribuya al desarrollo del sistema software abordado.

En contexto MDSE, este agente crea, supervisa y almacena modelos conformes a metamodelos proporcionados por el entorno, y los gestiona y transforma aplicando las herramientas disponibles. Para llevar a cabo estas tareas no se le requiere conocimiento experto en MDSE.

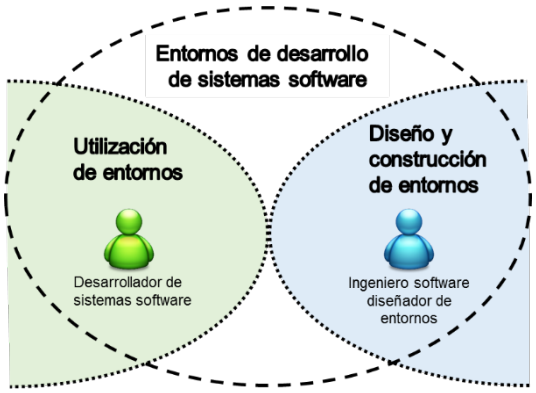


Figura 1.7 – Agentes relacionados con los entornos de desarrollo

- **Diseñador de entornos**. Ingeniero software experto en el diseño de procesos y entornos de desarrollo que hace avanzar las metodologías de trabajo en determinados dominios definiendo nuevas estrategias, herramientas y recursos. La construcción de un entorno en base a estas contribuciones le dota de la funcionalidad necesaria para dar soporte a los dominios que se desea cubrir.

  Este agente se ve a menudo abocado a solicitar la asistencia de expertos en la infraestructura de la plataforma que sirve de base a los entornos o incluso a hacerse experto en ella. Esta necesidad contrasta con su habitual reticencia a trabajar en dominios de conocimiento ajenos a su ámbito de experiencia, prefiriendo seguir haciendo uso de sus herramientas usuales que le permiten trabajar en su campo específico de forma aislada. Este conflicto supone un importante hándicap en la adopción de cualquier disciplina y sus tecnologías asociadas como base de los entornos.

La MDSE no es ajena a este hándicap. Por ejemplo, bajo un enfoque MDSE, los dominios conceptuales vienen formulados por lenguajes de modelado descritos mediante metamodelos y los procesos de desarrollo se realizan mediante herramientas que implementan transformaciones entre modelos. Así, además de especificar los modelos y metamodelos propios de los dominios a los que un entorno esté orientado, el agente diseñador de entornos tendría que desarrollar las transformaciones y demás herramientas, incluyendo su posterior mantenimiento frentes a posibles evoluciones de los metamodelos. Todo ello exigiría conocer en profundidad y hacer uso de tecnologías MDSE, en particular lenguajes mediante los que implementar transformaciones de modelos.

Así, la MDSE únicamente tendrá opción de ser aceptada y consolidarse como alternativa sobre la que diseñar entornos si a los ingenieros software (en general no expertos en tecnologías MDSE) se les brinda facilidad para realizar las tareas de desarrollo y actualización de las herramientas integradas en los entornos.

1.6.2 Objetivo global

El objetivo global que ha guiado el desarrollo de esta Tesis ha sido la búsqueda de estrategias y soluciones que incentiven la adopción de la disciplina MDSE por parte de los expertos encargados del diseño de entornos, fomentando de esta forma su implantación como base de los mismos. Por ello, se considera la incorporación de un tercer actor que aligere el cometido de tales expertos.

- **Desarrollador de infraestructura MDSE para entornos**. Su papel es construir una infraestructura (recursos y métodos) a modo de interfaz que al diseñador de entornos le facilite el acceso a los recursos genéricos de la tecnología MDSE, de forma que pueda trabajar utilizando tan sólo conceptos muy próximos a los dominios en los que es experto (metodologías, procesos y entornos) sin necesidad de hacerse también experto en las tecnologías MDSE. La Figura 1.8 muestra la incorporación de este tercer agente.

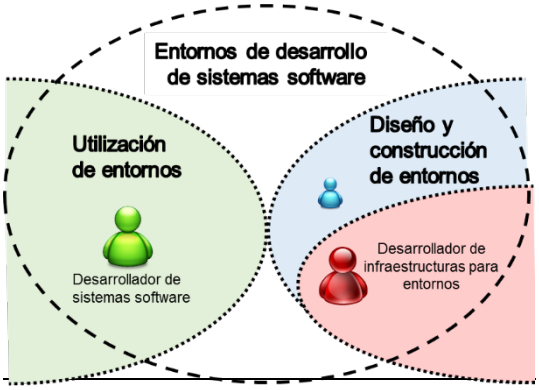


Figura 1.8 – Agente encargado del desarrollo de infraestructuras para entornos

La consecución de este objetivo central se ha abordado desde la perspectiva de realizar contribuciones en el campo de este tercer agente, de forma que se ha desglosado en los subobjetivos específicos que se exponen en detalle a continuación.

1.6.3 Objetivos concretos

Estudio de la gestión de la información en los entornos SDTRE.

Un entorno SDTRE puede verse como un repositorio de la información involucrada en las sucesivas etapas de los procesos de desarrollo de SDTREs. Definir y establecer estrategias para su gestión no es algo trivial pues no es algo que dependa únicamente de la naturaleza y características de ésta sino también de los distintos operadores humanos que, con diferentes puntos de vista, puedan interpretarla, o de las diversas herramientas que, con diferentes fines, puedan procesarla. Por ello, el primer objetivo concreto planteado ha sido el estudio de las propiedades de la información manejada en los procesos de desarrollo de SDTREs y en base a ellas, el análisis de alternativas para su representación, clasificación y organización en los entornos. En especial, en lo relativo a los posibles paradigmas de representación, se ha considerado esencial abordar el problema de la interoperabilidad entre ellos así como la búsqueda de estrategias adecuadas para una formalización amigable de la información de cara a los expertos de dominio.

Diseño de una metodología para producción de herramientas sin requerir conocimiento experto en tecnologías MDSE.

Sobre el objetivo general de facilitar la tarea del agente diseñador de entornos para fomentar la aceptación e implantación de la MDSE como base de los mismos, un objetivo fundamental planteado ha sido la exploración de técnicas que posibiliten la construcción de herramientas de gestión y procesado de modelos, así como la coevolución de las mismas frente a la evolución de los metamodelos que constituyen el ámbito conceptual de un entorno, sin requerir de los diseñadores conocimiento experto de las tecnologías MDSE subyacentes.

La definición del objetivo prevé el diseño de una metodología que permita construir herramientas genéricas, aplicables a modelos con independencia del metamodelo al que son conformes y, por tanto, fácilmente adaptables frente a la evolución de un metamodelo sin necesidad de intervención a bajo nivel de los diseñadores de entornos.

Propuesta de un modelo de referencia que formalice el diseño normalizado de entornos y reduzca el esfuerzo para su especificación.

La definición de este objetivo contempla proporcionar la capacidad de formular los procesos de desarrollo en forma de modelos, sin requerir desarrollar código de soporte basado en el conocimiento y uso de la infraestructura MDSE en la que se base el entorno.

Para ello, se han analizado las características comunes de los procesos de desarrollo MDSE, que básicamente consisten en la concatenación de tareas de procesamiento, refinamiento o transformación de los modelos así como de creación y supervisión de éstos por parte del operador. En base a este análisis, se propone una concepción de entornos basada en un modelo de referencia orientado a la interpretación de los modelos que definen los procesos de desarrollo soportados en un entorno.

1.7 Organización de la memoria

La exposición de las contribuciones se ha organizado en tres capítulos que se corresponden de forma casi directa con los diferentes objetivos que se han planteado para la Tesis. En los tres casos se complementan con implementaciones a modo de pruebas conceptuales realizadas para el entorno MAST de desarrollo de SDTREs.

Capítulo 2

El capítulo 2 analiza la naturaleza de la información contenida y manejada en los entornos SDTRE y en base a la caracterización obtenida aborda el problema de establecer estrategias idóneas para su gestión. En particular se analizan diversas opciones para su representación, en función del escenario (agentes, herramientas, objetivos, etc.) en que va a ser utilizada. Se analizan alternativas orientadas a que el acceso a los datos por parte de las herramientas resulte sencillo, a facilitar su interpretación por parte de los expertos del dominio al que ésta corresponde, a su persistencia e interoperatividad con otros entornos y a su escalabilidad cuando se pasa a entornos de ámbito industrial. Asimismo se tratan los mecanismos de conversión que posibilitan la interoperabilidad entre diferentes paradigmas de representación y el papel de pivote que juega MDSE con tal objetivo. En particular, el capítulo presta especial atención a establecer cuál de las alternativas analizadas ha de tomarse como representación de referencia y justificar la selección de una tecnología de soporte. También aborda en profundidad la elección de la mejor

solución para presentación de la información al experto de dominio y la necesidad de estrategias y formatos específicos para representar los modelos cuando traspasan el ámbito operativo del entorno.

En un marco MDSE, la formalización de un ámbito o dominio conceptual mediante metamodelado plantea un desafío esencial: la posibilidad y en su caso la conveniencia de formular metamodelos de rigurosidad exhaustiva, capaces de cubrir hasta el más mínimo detalle semántico del dominio en cuestión. En este capítulo se trata en profundidad este tema.

Por último, el capítulo también aborda y propone soluciones concretas al problema de la clasificación y organización de la información en los entornos.

Capítulo 3

El capítulo 3 analiza la naturaleza de la funcionalidad que puede ser abordada mediante herramientas genéricas, esto es, capaces de mantener su funcionalidad aún cuando cambien los metamodelos de los modelos sobre los que operan y, por tanto, aplicables sobre los modelos que motivan su diseño como sobre modelos conformes a metamodelos que serán añadidos en el futuro al entorno. Asimismo, se demuestra la relevancia y capacidad de las metaherramientas como forma de implementar las herramientas genéricas en un entorno y se muestra cómo pueden ser implementadas utilizando la técnica HOT.

Capítulo 4

El capítulo 4 propone una concepción genérica de entornos basados en MDSE, denominada MDDE (*Model-Driven Development Environment*), que facilita al experto diseñador de entornos su especificación e implementación. Ésta incluye la definición de un modelo de referencia que considera que el operador que utiliza un entorno lleva a cabo su actividad mediante la ejecución supervisada de procesos, la cual a su vez consiste en la ejecución secuencial o iterativa de operaciones más básicas denominadas tareas.

El modelo de referencia propuesto se describe desde un punto de vista estructural, definiendo los elementos conceptuales que lo constituyen y especificando para cada uno de ellos sus tipos derivados y su clasificación. Para posibilitar la formulación en forma de modelos tanto de los procesos como de los tipos de tareas instanciados en ellos, de forma que el diseño de un entorno consiste básicamente en la elaboración de modelos y no en el desarrollo de su código de implementación, se presenta un metamodelo de soporte. También se dedica una sección a definir la GUI normalizada que todo entorno acorde a esta concepción ha de presentar.

Por otro lado, en el capítulo se realiza un análisis de aspectos a los que ha de dar soporte funcional aquella plataforma que se desee emplear como base de entornos MDDE, justificando a través de pruebas de concepto la validez de Eclipse / EMF como plataforma de soporte.

Capítulo 5

Finalmente, el **capítulo 5** expone los resultados y las conclusiones del trabajo presentado en esta memoria de Tesis, junto a las líneas de trabajo futuro que surgen a partir de ella.

2 Representación y Gestión de la Información en los Entornos

Un entorno SDTRE es, entre otras cosas, un repositorio de la información generada y manejada en las sucesivas fases de los procesos de desarrollo de SDTREs, basado en un marco de referencia uniforme que facilite el acceso, el procesado y el mantenimiento de tal información.

Una primera característica de dicha información es su naturaleza poliédrica, esto es, se trata de información relativa a múltiples aspectos, como pueden ser especificaciones funcional y no funcional, diseños arquitectural y de implementación, verificaciones intermedias y finales, etc. Además, un mismo ítem de información ha de ser ofrecido con diferentes formatos en función de los distintos operadores humanos que, con diferentes puntos de vista, puedan interpretarlo o en función de las diversas herramientas que, con diferentes fines, puedan procesarlo. Sin embargo, aunque el ítem se describa de distintas maneras según el contexto, no deja de ser conceptualmente el mismo, por lo que entre sus diversas representaciones deben existir referencias que permitan mantener su consistencia. Por ello, una segunda característica es que se trata de información altamente interreferenciada entre sí.

Como se ha indicado en la sección 1.3, la disciplina MDSE proporciona recursos para abordar esta naturaleza poliédrica y altamente interreferenciada de la información contenida en un entorno SDTRE. Formulándola mediante diferentes modelos referenciados entre si, MDSE permite que cada ítem de información se pueda presentar de forma uniforme pero a su vez accesible desde diferentes puntos de vista. Además, si un entorno está basado en MDSE, no sólo establece el formato de codificación de los datos, sino también su estructura mediante metamodelos.

En muchas ocasiones, los modelos que se gestionan en un entorno son efímeros (meramente instrumentales) y sólo existen en el espacio de memoria de las aplicaciones, desapareciendo al finalizar su ejecución. En otras ocasiones, los modelos se crean para que persistan, y han de ser almacenados en repositorios persistentes (ficheros, BBDD, etc.).

Visión general del capítulo

En este capítulo se abordan aspectos relativos a la representación y gestión de la información en los entornos SDTRE.

La **sección 2.1 (*Espacios Tecnológicos e Interoperabilidad*)** trata acerca de la multiplicidad de formatos de representación de la información, en función de las herramientas con que va a ser manejada o del objetivo con que va a ser explotada. En concreto, esta sección se centra en los siguientes casos:

- **Como situación base, la información reside en el espacio de memoria de las herramientas**, con una formulación tal que el acceso a los datos desde sus códigos resulte fácil y con bajos tiempos de latencia. Si además las herramientas son genéricas, la información procesada ha de contener o referenciar aquella metainformación en que se basan sus diseños.
- **En otros casos, la información se formula para que sea fácil de interpretar por los expertos del dominio** al que ésta corresponde. Para ello se presenta sin referencias a la tecnología del entorno, expresándola mediante DSLs – habitualmente textuales – que resultan más amigables a tales expertos.

- **Cuando el objetivo es la persistencia de la información o la interoperatividad con otros entornos, ésta se formula mediante formatos estándar**, independientes tanto del entorno como de la plataforma que lo soporta.
- **Cuando el volumen de información crece, tanto en términos de cantidad de modelos como de su tamaño** (número de elementos de modelo contenidos)**, se requiere su almacenamiento en BBDD**. En este caso, la información debe ser formulada mediante una estrategia acorde a su naturaleza (relacional, NoSQL, OO, orientada a grafos, etc.).

En esta sección también se aborda la interoperabilidad entre los diferentes paradigmas de representación, los mecanismos de conversión entre ellos y el papel de pivote que juega MDSE con tal objetivo.

La **sección 2.2 (*Modelware como TS base para el manejo de la información*)** establece que el espacio tecnológico que debe tomarse como fundamental cuando se desea aplicar una metodología conducida por modelos es lógicamente *Modelware* , y se presenta su rol de pivote para interconectar otros espacios.

La **sección 2.3 (*Laxitud de los MetaModelos y Especificación de Restricciones*)** aborda la conveniencia de utilizar metamodelos laxos que, aunque no garanticen de forma completa la coherencia de los datos contenidos en los modelos, sí facilitan la utilización de éstos últimos por diferentes herramientas, reduciendo la necesidad de replicar modelos con una información en gran parte compartida. Es común que se requiera incorporar metainformación complementaria en forma de restricciones, que interpretada por herramientas genéricas del entorno garanticen la coherencia de los datos.

La **sección 2.4 (*Formalización Amigable de la Información para el Experto de Dominio*)** aborda la construcción y utilización de DSLs para presentar la información de forma que resulte amigable a los expertos humanos que la han de interpretar.

La **sección 2.5 (*Persistencia e Intercambio de Modelos*)** aborda la necesidad de estrategias y formatos específicos para representar los modelos cuando traspasan el ámbito operativo del entorno. Esto ocurre cuando:

- Se almacenan los modelos en un repositorio persistente.
- Se intercambia la información con otro entorno.
- En el caso de un entorno distribuido, los modelos deben transferirse a través de una red de comunicaciones.

La **sección 2.6 (*Clasificación y Organización de la Información en los Entornos*)** aborda estrategias de escalabilidad de los entornos. Los modelos son simples unidades de información que se crean como parte de una tarea de supervisión o procesamiento, pero en un entorno de escala industrial pueden coexistir miles de modelos, tanto versiones diferentes del sistema que se desarrolla como modelos de naturaleza completamente distinta. En estos casos, los modelos se organizan en sistemas de ficheros, espacios de trabajo o BBDD.

Por último, la **sección 2.7 (*Escenario MAST-2*)** se apoya en MAST-2 para proporcionar ejemplos que ilustran los aspectos tratados en este capítulo acerca de la representación y gestión de la información en los entornos SDTRE.

2.1 Espacios Tecnológicos e Interoperabilidad

2.1.1 Noción de Espacio Tecnológico

La noción de *Espacio Tecnológico*[36] (TS) fue inicialmente introducida en [93] dentro de un marco sobre discusión de problemas de conexión entre tecnologías diferentes y posteriormente desarrollada más en profundidad en [94, 95]. En principio, en [93] se propone la siguiente descripción informal:

> *Un TS es un contexto operativo con un conjunto de conceptos asociados, cuerpo de conocimiento, herramientas, destrezas requeridas y posibilidades, asociado a menudo a una cierta comunidad de usuarios, con know-how compartido, soporte educativo, literatura común e incluso workshops y congresos regulares. Se trata de un área de experiencia e investigación y un repositorio de recursos tanto concretos como abstractos.*

El término se introduce con la intención de abstraer a las tecnologías y poder razonar acerca de sus similitudes, diferencias y posibilidades de integración. La formulación de las características comunes con un concepto genérico proporciona un marco comparativo que facilita la comprensión de las diferentes tecnologías.

2.1.2 Espacios tecnológicos

La Figura 2.1 muestra los diversos espacios tecnológicos que se suelen citar en la bibliografía. Se muestran en azul los que se tratan en esta Tesis.

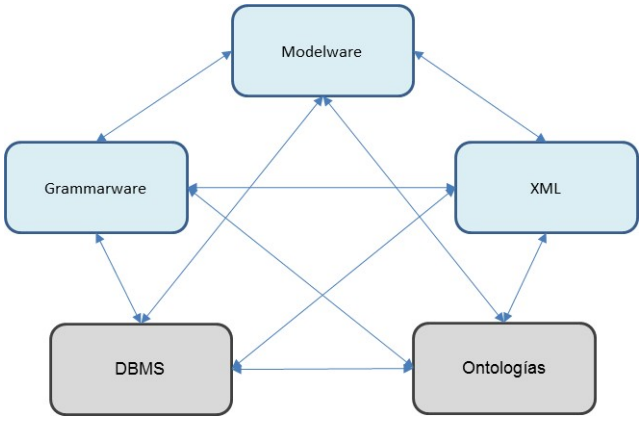


Figura 2.1 – Espacios Tecnológicos típicos

En general, los espacios tecnológicos son tecnologías alternativas para la gestión de modelos. Su análisis comparado revela que en muchos casos se organizan de forma semejante a la arquitectura de 3+1 niveles definida explícitamente en *Modelware*. A continuación se expone la naturaleza y justificación de los espacios mostrados en la Figura 2.1:

- ***Modelware*** [96]. La información relativa a un sistema se representa mediante modelos conformes a metamodelos que formalizan los conceptos de un determinado dominio. A su vez, los metamodelos se construyen a partir de las primitivas especificadas por un

[36] También *Espacio Técnico*, dependiendo del autor.

metametamodelo único, propio del *framework* MDSE concreto elegido para dar soporte al espacio. Por ejemplo, el metametamodelo Ecore proporcionado por Eclipse/EMF.

La principal función de este TS es describir conceptualmente la forma en que los modelos de los sistemas se representan en la memoria de las aplicaciones de un entorno SDTRE.

- ***Grammarware***. En este TS los modelos se formulan como documentos textuales en base a un lenguaje formal cuya sintaxis está especificada mediante una gramática, generalmente de tipo EBNF, que recoge los conceptos de dominio. Aquí se pueden identificar claramente los tradicionales metaniveles, en forma de documento textual, gramática y especificación EBNF.

  La principal contribución de este TS es facilitar al experto de dominio la comprensión y expresión de la información, buscando que le resulte natural y amigable, y sin requerirle que posea conocimientos de la tecnología MDSE que da soporte al entorno. Así, en la concepción de entorno SDTRE que se plantea en esta Tesis, el TS *Grammarware* se utiliza en las interfaces de interacción entre el entorno y los expertos de dominio.

- **XML**. En este TS la información se formula en XML, como forma estandarizada y universal de representación, compatible con todas las plataformas y sistemas.

  El concepto básico en este TS es la formulación de los modelos como un flujo (*stream*) de caracteres sobre el que se aplican tanto restricciones de *well-formedness*, definidas por las reglas de la gramática XML, como restricciones de validez definidas a través de documentos W3C-Schema.

  La principal contribución de este TS es proporcionar un mecanismo universal – independiente del dominio – para codificar los modelos, posibilitando que traspasen el ámbito operativo de un entorno cuando:

  - Se han de almacenar en un repositorio para su persistencia.
  - Se desea su intercambio entre entornos basados en diferentes lenguajes o plataformas donde la representación binaria de la información es diferente.
  - Se transfieren por una red de comunicación estándar, en el caso de que el entorno sea distribuido.

  En este TS, las referencias cruzadas entre modelos deben formalizarse mediante identificadores textuales (URI).

- **Sistemas de BBDD (DBMS)**. En este TS la información se formula mediante estructuras (tablas, objetos, etc.) que sean compatibles con la tecnología de BBDD utilizada.

  La principal función de este TS en un entorno es proporcionar escalabilidad si la información incluida en un modelo o en un conjunto de modelos es grande, por lo que su inclusión es necesaria cuando los entornos pasan de prototipo a un uso industrial. Asimismo, es necesario cuando los modelos deben ser compartidos concurrentemente por diferentes entornos y se hace necesario garantizar la seguridad transaccional en el acceso a la información.

- **Ontologías**. Otro ejemplo de TS conforme a la organización de metacapas es el de Ontologías o Representación del Conocimiento [97, 98], en el cual los niveles M1, M2 y M3 se corresponden respectivamente con *hechos*, *ontologías* y *ontología de nivel superior*.

  Este TS se utiliza principalmente cuando el entorno incorpora herramientas basadas en principios de inteligencia artificial.

2.1.3 Interoperabilidad entre Espacios Tecnológicos

Los espacios tecnológicos deben ser vistos como escenarios de trabajo complementarios que facilitan implementar una funcionalidad. De hecho, la transformación de modelos de un espacio a otro es la mejor o más sencilla forma de llevar a cabo muchas de las tareas de desarrollo en un entorno. Así, algunas operaciones pueden ser llevadas a cabo en un TS y el resultado ser transferido a otro. Es más, en ocasiones, moverse a otro TS para resolver un problema es parte esencial de la solución.

Un entorno debe proporcionar mecanismos que comuniquen los espacios y permitan su interoperabilidad, de forma que para el operador y las herramientas sea uniforme el uso de los modelos con independencia del TS en que estén formulados. La Figura 2.1 también ilustra esta perspectiva, mostrando los diversos espacios interconectados.

Interoperabilidad

IEEE define formalmente la interoperabilidad entre herramientas o entre sistemas (o subsistemas/componentes de un mismo sistema) como su *habilidad para intercambiar información y utilizar tal información intercambiada*.

Son numerosos los escenarios donde surgen necesidades de interoperabilidad, como por ejemplo entre herramientas consecutivas de un proceso de desarrollo en cadena, en un contexto de trabajo colaborativo donde varios agentes necesiten trabajar conjuntamente en la misma tarea usando diferentes herramientas, en el ámbito de la integración de sistemas, en temas de evolución de lenguajes y herramientas al buscar retrocompatibilidad con versiones previas, etc.

La interoperabilidad es en general un campo complicado que requiere tratar tanto aspectos sintácticos como semánticos. Cada sistema o herramienta puede usar un formato sintáctico diferente y, asimismo, que interprete internamente su semántica de forma específica.

- Mientras que en el plano semántico la interoperabilidad se define a nivel de sistemas concretos,
- la interoperabilidad sintáctica está directamente ligada a la tecnología concreta subyacente bajo cada sistema o herramienta, es decir, el TS de base. Por tanto, esta interoperabilidad se define genéricamente a nivel de TS.

Puentes de interoperabilidad

La estrategia adecuada para abordar la interoperatividad es proveer conexiones genéricas entre las partes involucradas, independientemente del proyecto o contexto específico en que se apliquen. Proporcionar una solución artesanal no es recomendable pues es una tarea proclive a error, costosa y efímera.

La Figura 2.2 ilustra la formalización de un mismo dominio en distintos espacios (los tres considerados básicos en el contexto de esta Tesis) y la sincronización entre ellos a través de *puentes de interoperabilidad*, tanto a nivel M1 como a nivel M2.

- Los puentes de interoperabilidad a nivel M2 conllevan establecer correspondencias entre las formalizaciones de los conceptos de dominio en cada TS, de forma que sean automatizables por herramientas genéricas.

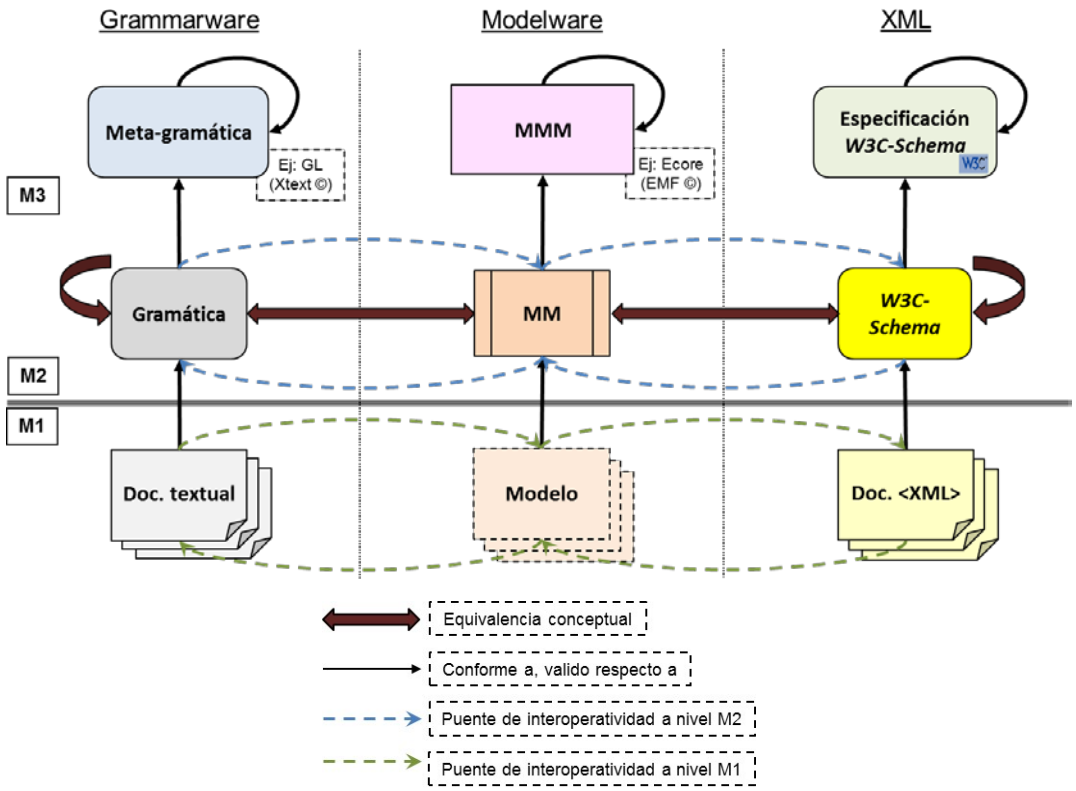


Figura 2.2 – Formalización de un dominio en distintos espacios tecnológicos

- A nivel M1, los puentes de interoperabilidad suponen mecanismos que realizan la conversión automática entre los artefactos (documentos textuales, modelos, documentos XML) que representan la información del sistema en los diferentes espacios. Las correspondencias entre las formalizaciones conceptuales en el nivel M2 hacen posible que estas pasarelas a nivel M1 sean soportadas por herramientas genéricas, independientes de los dominios.

2.2 *Modelware* como TS base para el manejo de la información

2.2.1 Conveniencia de *Modelware*

A pesar de que todos los espacios tecnológicos puedan ser percibidos como tecnologías de gestión de modelos, en el sentido genérico de representaciones abstractas de sistemas del mundo real, el TS que debe tomarse como fundamental cuando se desea aplicar una metodología conducida por modelos es *Modelware*. Por un lado, es el TS propio de MDSE, constituido por artefactos que son directamente las nociones de modelo y metamodelo, independientemente de cualquier notación o representación concreta, y como tal su aplicación tiene como fin último la representación de los modelos en el espacio de memoria de las aplicaciones. Por otro lado, el soporte a transformaciones entre modelos (las operaciones básicas en MDSE) se encuentra muy consolidado en este TS, frente a su desarrollo mediante XSLT en XML o mediante tratamiento de texto plano en *Grammarware*. Esto, unido al hecho de que EMP, la tecnología MDSE dominante en la actualidad, integre a otros espacios proporcionando interoperabilidad con *Modelware*, hace que éste último se erija como pivote en muchos escenarios, afianzándose por tanto como TS base para el manejo de la información.

La Figura 2.3 toma como base la Figura 2.1 pero ilustra el papel de pivote de *Modelware*, prescindiendo de puentes directos entre el resto de espacios.

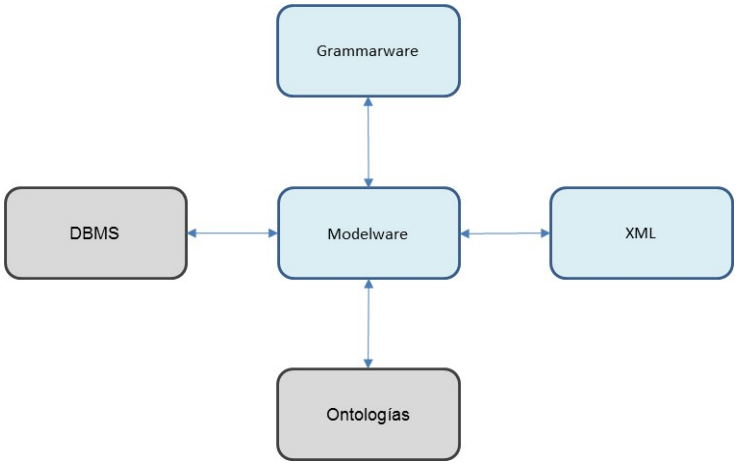


Figura 2.3 – Espacios Tecnológicos típicos interconectados mediante *Modelware*

Interoperabilidad Dirigida por Modelos

En la mayoría de los escenarios no sólo es necesario interconectar sintácticamente los espacios correspondientes a los sistemas o herramientas del entorno, sino que también se requiere implementar su interoperabilidad en el plano semántico. MDSE posibilita abordar este problema a un alto nivel de abstracción, mediante la aplicación de técnicas *model-driven.* Esto es lo que se denomina Interoperabilidad Dirigida por Modelos (MDI) [99], que en esencia, consiste en un proceso compuesto por tres fases:

1. **Formular el esquema interno de cada sistema**, tanto en su propio TS como en *Modelware*, estableciendo *proyectores* (conversores sintácticos) entre ellos.

   Por ejemplo, en caso de que el sistema almacene datos XML, el esquema interno sería el correspondiente *W3C-Schema*, mientras que si gestiona información en una BBDD relacional, se trataría del correspondiente *esquema relacional*. En *Modelware*, el esquema interno adopta la forma de metamodelo.

2. **Alinear los metamodelos** emparejando conceptos relacionados, es decir, especificar la correspondencia semántica.

3. **Explotar esa información semántica mediante transformaciones M2M** para portar datos (modelos) conformes al metamodelo del primer sistema a modelos conformes al metamodelo del segundo.

La Figura 2.4 muestra cómo aborda MDI la consecución de interoperabilidad completa entre dos sistemas/herramientas A y B, basados respectivamente en los espacios *Grammarware* y XML.

Tomando por ejemplo el sentido *Grammarware* → XML, el puente MDI se puede descomponer básicamente en tres partes:

1. **Pasarela sintáctica** *Grammarware* → *Modelware* (*proyección inyectora* o simplemente *inyección*).
2. **Correspondencia semántica** A → B en el seno de *Modelware*.
3. **Pasarela sintáctica** *Modelware* → XML (*proyección extractora* o simplemente *extracción*).

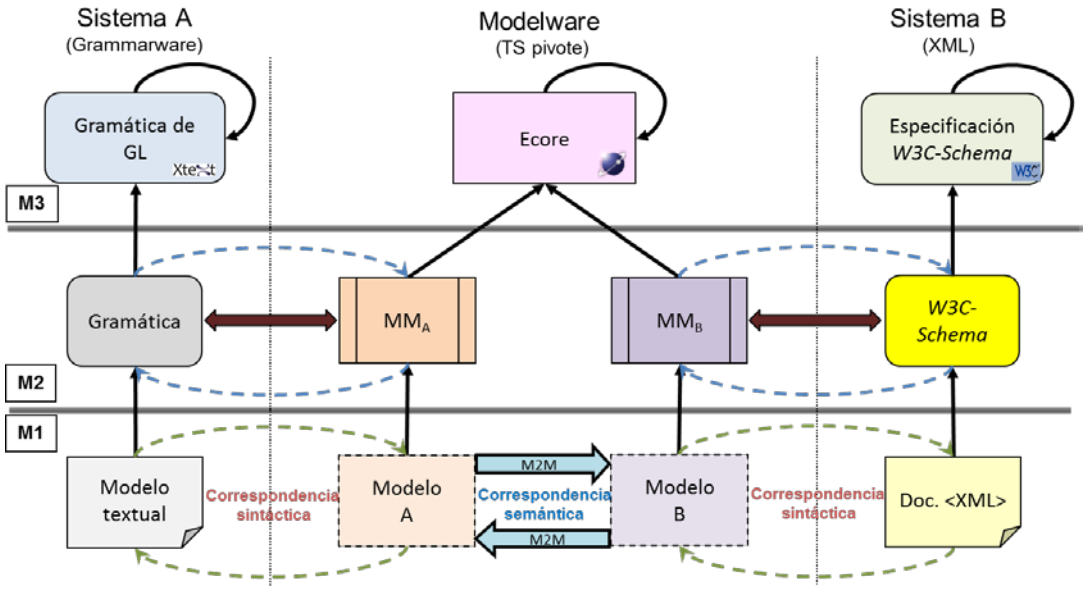


Figura 2.4 – Puente MDI entre sistemas/herramientas basados en *Grammarware* y XML

Tomando por ejemplo el sentido *Grammarware* → XML, el puente MDI se puede descomponer básicamente en tres partes:

4. **Pasarela sintáctica** *Grammarware* → *Modelware* (*proyección inyectora* o simplemente *inyección*).
5. **Correspondencia semántica** A → B en el seno de *Modelware*.
6. **Pasarela sintáctica** *Modelware* → XML (*proyección extractora* o simplemente *extracción*).

Como puede observarse, las correspondencias sintácticas nada tienen que ver con la semántica conceptual de los sistemas implicados, sino que son relativas a espacios tecnológicos, representando la comunicación de *Modelware* tanto con *Grammarware* como con XML. Además de la transformación M2M que implementa la correspondencia semántica, son clave los proyectores, los cuales pueden ser directamente codificados usando un GPL o preferiblemente implementados mediante transformaciones M2T o T2M.

2.2.2 Selección de tecnología MDSE de soporte: EMP

En el capítulo 1 se indicó la elección en esta Tesis de la tecnología desarrollada en el proyecto EMP, lo que conduce a la selección de la tecnología EMF como base de *Modelware*, de la tecnología Xtext como base de *Grammarware* y la tecnología de las especificaciones XMI de OMG y Schema de W3C como base del espacio *XML*.

La Figura 2.5 muestra los espacios *Grammarware* y *Modelware* (soportados respectivamente por los *frameworks* Xtext y EMF) conectados por puentes de interoperabilidad tanto a nivel M2 como a nivel M1, puentes proporcionados conjuntamente por ambos *frameworks*.

En la capa M2, Xtext ofrece la posibilidad de crear un proyecto Xtext a partir de un modelo Ecore ya existente, generando una gramática completa y totalmente funcional. Simétricamente, permite desarrollar una gramática desde cero y a partir suyo inferir el correspondiente metamodelo. En lo relativo al nivel M1, el API conjunto de EMF y Xtext permite implementar la conversión entre modelos textuales y modelos EMF.

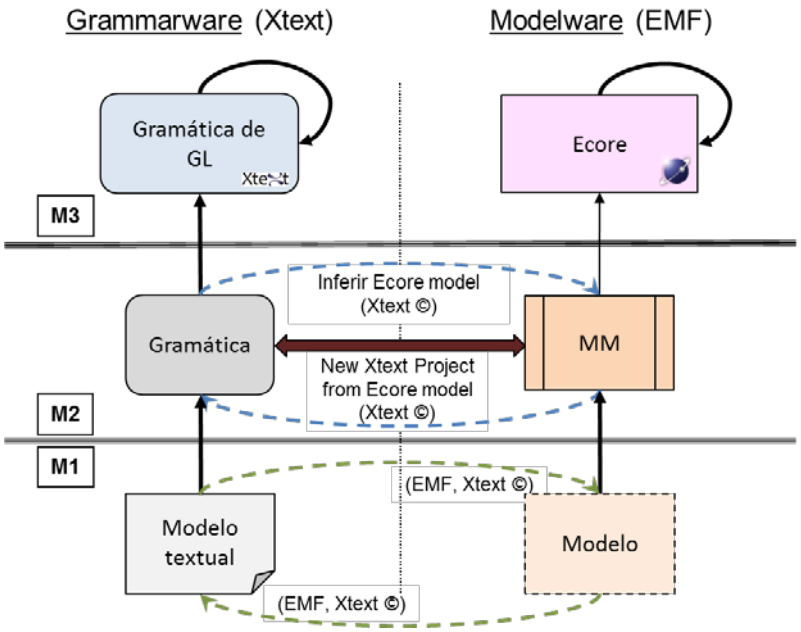


Figura 2.5 – Interoperabilidad entre los espacios *Grammarware* (Xtext) y *Modelware* (EMF)

De forma análoga, el propio EMF proporciona puentes entre *Modelware* y XML (Figura 2.6).

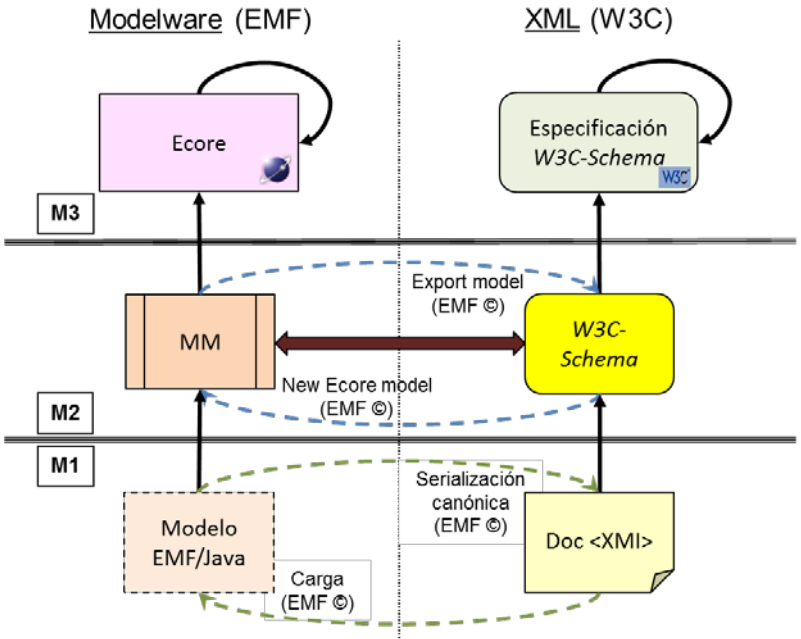


Figura 2.6 – Interoperabilidad entre los espacios *Modelware* (EMF) y XML

Por un lado, el asistente de creación de un proyecto EMF permite elegir como punto de partida un W3C-Schema y obtener a partir suyo el modelo Ecore correspondiente y por otro, es posible exportar a W3C-Schema un metamodelo ya construido. Lo interesante respecto a estos puentes a nivel M2 proporcionados por EMF es que conllevan una interoperabilidad inmediata a nivel M1. Por ejemplo, el puente a nivel M2 en sentido *Modelware* → XML produce, a partir de un metamodelo, un W3C-Schema con una morfología muy particular, la cual permite que un modelo conforme al metamodelo pueda ser cargado a partir de o almacenado en forma de un documento XML válido frente al W3C-Schema producido.

2.3 Laxitud de los Metamodelos y Especificación de Restricciones

2.3.1 Formulación laxa de metamodelos

Desde un punto de vista conceptual ortodoxo, la estructura de la información incluida en un metamodelo debería garantizar que los modelos conformes a él sean siempre coherentes respecto a la semántica del dominio conceptual cubierto. De no ser así, los modelos serán conformes al metamodelo, pero podrían no ser válidos para ciertos usos o incluso bajo ninguna circunstancia. Sin embargo, es algo común formular metamodelos de forma laxa, esto es, los modelos conformes admiten toda la información que requiere el dominio cubierto, pero no está garantizado que sean siempre coherentes y libres de errores semánticos.

Algunas razones por las que puede resultar necesario o conveniente definir metamodelos laxos son las siguientes:

- **En ocasiones resulta simplemente imposible modelar un dominio conceptual de manera absolutamente rigurosa**. Los lenguajes de metamodelado, buscando una curva de aprendizaje de baja pendiente, ofrecen un conjunto reducido de construcciones para definir metamodelos. Con ello, logran la facilidad de uso a cambio de ser únicamente capaces de expresar la estructura de los metamodelos, incluyendo la definición de restricciones muy básicas (por ejemplo, de cardinalidad).
- **Preservar tan simple como sea posible la estructura del metamodelo, con el propósito de facilitar su mantenimiento y el desarrollo de futuras extensiones**. Frente a casos en que resulta imposible que la formulación del metamodelo sea exhaustiva, en otros escenarios puede ser posible diseñar el metamodelo de forma que asegure la coherencia de los modelos conformes a él, pero requiriendo para ello una estructura interna extremadamente compleja, caracterizada por un gran número de tipos primitivos en lugar de los usuales (int, real, boolean, char, string, etc.) y una jerarquía muy profunda de herencia de clases, tratando de especializar al máximo las posibles asociaciones y sus multiplicidades.
- **Querer utilizar un único metamodelo para formalizar modelos que, como consecuencia de que se usan en diferentes procesos, han de satisfacer diferentes reglas o condiciones**. Por ejemplo, en el caso de que diferentes herramientas de un entorno puedan requerir características específicas de los modelos, podría ser preferible utilizar un metamodelo único acorde a la naturaleza nuclear del dominio descrito, en lugar de definir un metamodelo especializado para cada uso particular de los modelos.

2.3.2 Especificación de restricciones

La formulación laxa de un metamodelo se traduce en la existencia de un conjunto de puntos (laxitudes) en su estructura donde los modelos conformes a él pueden presentar incoherencias en relación a la semántica del dominio conceptual cubierto. Para prevenir este problema, la formulación laxa puede complementarse definiendo un conjunto de reglas adicionales, llamadas *restricciones de integridad*, que hacen referencia a los elementos del metamodelo y que deben ser cumplidas por cualquier modelo conforme a él para poder ser considerado coherente, independientemente del proceso en que vaya a ser utilizado. Cuando dichas reglas han sido definidas, el metamodelo ha sido enriquecido mediante la especificación de tales restricciones de integridad, las cuales también podrían denominarse *restricciones intrínsecas de coherencia*.

Distinto es el caso de aquellas restricciones impuestas por las diferentes herramientas de un entorno para poder actuar sobre los modelos. Obviamente, toda herramienta que procesa un determinado tipo de modelos (conformes a un cierto metamodelo), debe demandar que se trate de modelos coherentes, válidos respecto a aquellas restricciones intrínsecas que hayan podido ser especificadas sobre el metamodelo, no garantizando en caso contrario la corrección de los resultados producidos. Adicionalmente, una herramienta puede definir *restricciones específicas* que los modelos han de satisfacer para poder ser procesados por ella. Los motivos pueden ser muy diversos, como por ejemplo que la herramienta sólo sea aplicable a algún tipo limitado de modelos debido a que aún se encuentra en una fase preliminar de su desarrollo.

Cuando sobre un metamodelo se define un conjunto de restricciones, tanto si son intrínsecas como específicas de herramienta, los modelos conformes a él que además satisfacen las restricciones son *válidos*, bien desde un punto de vista de coherencia o para un propósito determinado. La validez de los modelos es una propiedad que ha de ser certificada por alguna herramienta desarrollada en base a tales restricciones.

Lenguajes de formulación de restricciones: OCL

OMG ha definido el lenguaje estándar OCL (*Object Constraint Language*) [100, 101] para, entre otras cosas, la formulación de restricciones sobre un metamodelo. Como estándar, su adopción facilita la interoperabilidad entre herramientas de diferentes orígenes diseñadas para trabajar sobre metamodelos con restricciones. Originalmente, OCL evolucionó como parte de UML, donde se utiliza para los aspectos matemáticos detallados que no son apropiados para exposición gráfica. Posteriormente, OCL 2.0 fue separado de UML 2.0 en reconocimiento a su gran utilidad. Asimismo, OCL ha sido reutilizado en los estándares MOFM2T y QVT como base para especificar las reglas de transformaciones de modelos.

En cuanto a tipos de restricciones, OCL distingue básicamente dos tipos: los invariantes, que son aserciones que deben ser siempre verdaderas, y otras restricciones que deben ser ciertas sólo en un determinado punto temporal, como las precondiciones y postcondiciones impuestas sobre una operación. En esta Tesis sólo se consideran las restricciones invariantes. Un invariante se define en el contexto de una clase y se compone de un nombre y una expresión booleana que ha de ser cierta para cada instancia de dicha clase. La especificación de un invariante OCL tiene una formulación con la siguiente estructura:

```
context Nombre_Clase inv Nombre_Restricción:
   Expresión OCL booleana
```

2.3.3 Metamodelo MAST-2: Ejemplo de metamodelo laxo y uso de restricciones de integridad

Cuando se ha abordado el diseño del metamodelo MAST-2 con el que dar soporte a la formulación de modelos de comportamiento temporal de SDTREs, se han considerado dos requisitos importantes:

- **Sencillez de uso** cuando sea empleado como base sobre la que construir los modelos de comportamiento temporal de los SDTREs que se diseñen.
- **Facilidad de extensión** cuando los expertos en software de tiempo real deseen incorporar nuevos aspectos y métodos de análisis y diseño de SDTREs.

Por ello, el metamodelo MAST-2 se ha definido laxo, y en consecuencia se ha especificado sobre él un conjunto de restricciones de integridad, denominadas *Restricciones MAST-2*. Además, también se han especificado diversos conjuntos de restricciones adicionales, específicas a las herramientas disponibles en el entorno.

> ☞ La relación completa y detallada de las laxitudes exhibidas por el metamodelo MAST-2 así como del conjunto de restricciones de integridad especificadas sobre él (*Restricciones MAST-2*) puede encontrarse en http://www.istr.unican.es/members/cesarcuevas/phd/artifactsMAST2.html
>
> Se listan clasificadas en función de si la laxitud está localizada en un atributo, en un conjunto de atributos (de la misma clase), en una referencia o en un conjunto de referencias.

Con el propósito de ilustrar los conceptos expuestos en esta sección, los apartados a continuación presentan algunos ejemplos de laxitudes existentes en el metamodelo MAST-2, junto a la restricción de integridad correspondiente.

Laxitudes localizadas en un atributo

En esta categoría se encuadran únicamente 2 laxitudes, ambas del mismo tipo: Una clase define un atributo numérico cuya semántica se corresponde con un subrango de valores dentro del rango completo cubierto por su tipo. Sin embargo, nada en el metamodelo fuerza que el valor asignado al atributo en las instancias de dicha clase presentes en modelos MAST-2 pertenezca al subrango válido.

Ejemplo: el metamodelo MAST-2 define la clase *Processing_Resource* para modelar el concepto de componente hardware que lleva a cabo actividades del sistema modelado. Tal y como muestra la Figura 2.7, la clase formula el atributo Speed_Factor para modelar la capacidad de procesamiento.

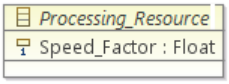


Figura 2.7 – Clase *Processing_Resource*

Evidentemente, en cualquier instancia de esta clase se ha de cumplir que:

```
Speed_Factor > 0.0
```

Sin embargo, nada en el metamodelo asegura que el valor asignado respete dicha condición. Por tanto, se define un invariante, identificado ***i_1_1_a***, cuya formulación OCL es mostrada por el Código 2.1.

```
context Processing_Resource inv i_1_1_a:
   Speed_Factor > 0.0
```

Código 2.1 – Formulación OCL del invariante *i_1_1_a*

Laxitudes localizadas en un conjunto de atributos (de la misma clase)

En esta categoría se encuadran un total de 23 laxitudes, todas del mismo tipo: Una clase define un conjunto de atributos de tipo numérico los cuales poseen una semántica que impone un cierto orden relacional a los valores que se les asignen. Sin embargo, a pesar de que el nombre de los atributos pueda sugerir tal orden, nada en el metamodelo fuerza que ese orden sea respetado en las instancias de dicha clase presentes en modelos MAST-2.

Ejemplo: El metamodelo MAST-2 define la clase *Timer* para modelar el concepto de temporizador hardware. Tal y como muestra la Figura 2.8, la clase formula los atributos Max_Overhead, Avg_Overhead y Min_Overhead para modelar el *overhead* asociado con la gestión de un temporizador.

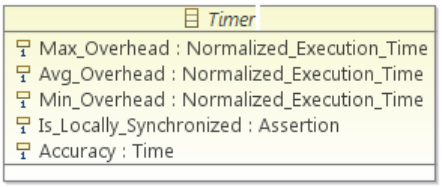


Figura 2.8 – Clase *Timer*

Evidentemente, en cualquier instancia de esta clase se ha de cumplir que:

$$Max\_Overhead \geq Avg\_Overhead \geq Min\_Overhead$$

Sin embargo, nada en el metamodelo asegura que los valores asignados respeten dicho orden. Por tanto, se define un invariante, identificado ***i_2_1_a***, cuya formulación OCL es mostrada por el Código 2.2.

```
context Timer inv i_2_1_a:
   Max_Overhead >= Avg_Overhead and Avg_Overhead >= Min_Overhead
```

Código 2.2 – Formulación OCL del invariante *i_2_1_a*

Laxitudes localizadas en una referencia

En esta categoría se encuadran un total de 30 laxitudes, de diversa naturaleza. Uno de tales escenarios consiste en que una clase abstracta posee una referencia cuyo tipo es otra clase abstracta pero entre sus respectivas subclases existen incompatibilidades. Sin embargo, nada en el metamodelo fuerza que las instancias de dichas clases presentes en modelos MAST-2 se asocien de manera compatible.

Ejemplo: El metamodelo MAST-2 define las clases abstractas *Schedulable_Resource* y *Scheduling_Parameters* para modelar respectivamente los conceptos de entidad planificable en un recurso de procesamiento (unidad de ejecución concurrente en un procesador o red de comunicaciones) y los parámetros que se adjuntan a cada uno de tales recursos planificables y que son usados por su planificador para tomar decisiones de planificación cuando éstos compiten entre si.

La Figura 2.9 muestra cómo se relacionan ambas clases y qué subclases las extienden. Tal y como puede observarse, la clase *Schedulable_Resource* define la referencia Scheduling_Parameters de tipo *Scheduling_Parameters*.

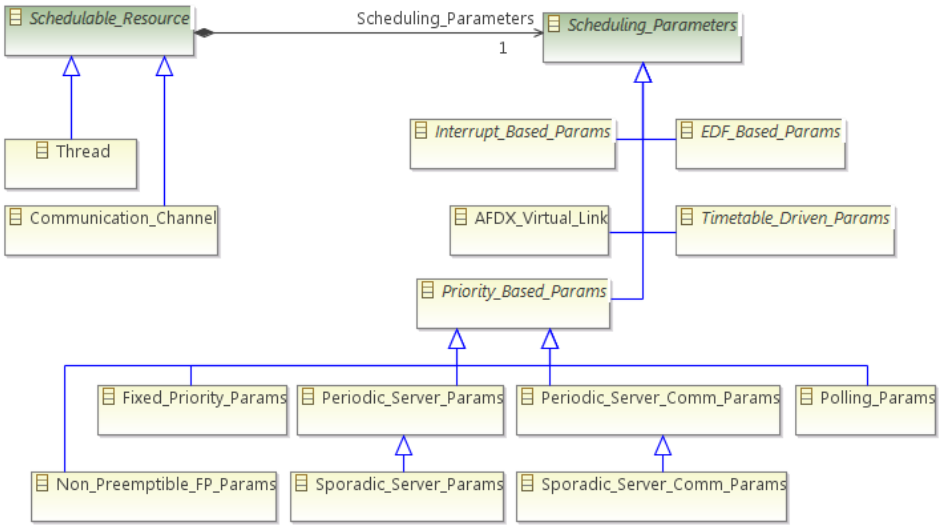


Figura 2.9 – Relación entre las clases *Schedulable_Resource* y *Scheduling_Parameters*

Dada cualquier pareja de instancias de estas clases, ha de respetarse la compatibilidad recogida en la Tabla 2-1.

Tabla 2-1 – Compatibilidad entre recursos planificables y parámetros de planificación

Schedulable_Resource	Scheduling_Parameters
Thread	*Interrupt_Based_Params* Periodic_Server_Params (Sporadic_Server_Params)
	Non_Preemptible_FP_Params Fixed_Priority_Params Polling_Params *EDF_Based_Params* *Timetable_Driven_Params*
Communication_Channel	Periodic_Server_Comm_Params (Sporadic_Server_Comm_Params) ADFX_Virtual_Link

Sin embargo, nada en el metamodelo asegura que las instancias asociadas respeten dicha compatibilidad, de forma que un recurso planificable podría contener parámetros de planificación incompatibles con su naturaleza. Por tanto, se definen dos invariantes, identificados ***i_3_5_a_I*** e ***i_3_5_a_II***:, cuyas formulaciones OCL son mostradas por el Código 2.3 y el Código 2.4.

```
context Thread inv i_3_5_a_I:
   Scheduling_Parameters.oclIsKindOf(Interrupt_Based_Params) or
   Scheduling_Parameters.oclIsTypeOf(Non_Preemptible_FP_Params) or
   Scheduling_Parameters.oclIsTypeOf(Fixed_Priority_Params) or
   Scheduling_Parameters.oclIsTypeOf(Polling_Params) or
   Scheduling_Parameters.oclIsKindOf(Periodic_Server_Params) or
   Scheduling_Parameters.oclIsKindOf(EDF_Based_Params) or
   Scheduling_Parameters.oclIsKindOf(Timetable_Driven_Params)
```

Código 2.3 – Formulación OCL del invariante i_3_5_a_I

```
context Communication_Channel inv i_3_5_a_II:
   Scheduling_Parameters.oclIsTypeOf(Non_Preemptible_FP_Params) or
   Scheduling_Parameters.oclIsTypeOf(Fixed_Priority_Params) or
   Scheduling_Parameters.oclIsTypeOf(Polling_Params) or
   Scheduling_Parameters.oclIsKindOf(Periodic_Server_Comm_Params) or
   Scheduling_Parameters.oclIsKindOf(EDF_Based_Params) or
   Scheduling_Parameters.oclIsKindOf(Timetable_Driven_Params) or
   Scheduling_Parameters.oclIsTypeOf(AFDX_Virtual_Link)
```

Código 2.4 – Formulación OCL del invariante *i_3_5_a_II*

Laxitudes localizadas en un conjunto de referencias

En esta categoría se encuadran un total de 15 laxitudes, de diversa naturaleza. Algunos de tales escenarios son:

a. De una clase parten dos cadenas de referencias que desembocan en sendas clases abstractas pero entre sus respectivas subclases existen incompatibilidades. Sin embargo, nada en el metamodelo fuerza que las instancias de dichas clases presentes en modelos MAST-2 se asocien (en este caso indirectamente) de manera compatible.

   Ejemplo: El metamodelo MAST-2 define la clase `Step` para modelar el concepto de *actividad*, esto es, la ejecución de una operación (ejecución de código o transmisión de mensaje) por parte de un recurso planificable en un recurso de procesamiento dado. Tal y como muestra la Figura 2.10, dicha clase formula sendas referencias `Step_Operation` y `Step_Schedulable_Resource` para designar a las instancias implicadas de las clases *Schedulable_Resource* y *Operation*.

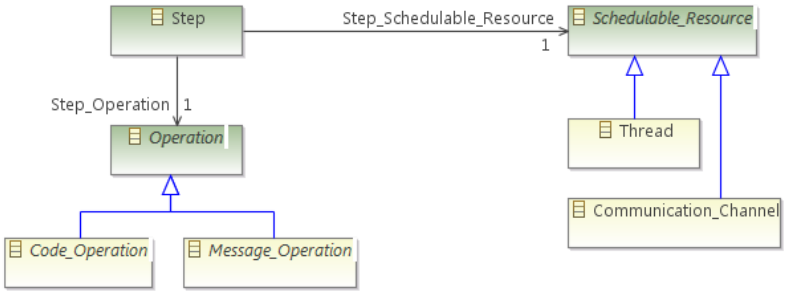


Figura 2.10 – Clase `Step` vinculando las clases *`Operation`* y *`Schedulable_Resource`*

   Dada cualquier instancia de la clase `Step`, ha de respetarse la compatibilidad recogida en la Tabla 2-2.

Tabla 2-2 – Compatibilidad entre recursos planificables y operaciones

Schedulable_Resource	Operation
Thread	*Code_Operation*
Communication_Channel	*Message_Operation*

   Sin embargo, nada en el metamodelo asegura que las instancias asociadas respeten dicha compatibilidad, de forma que una actividad podría estar vinculando una pareja imposible de operación / recurso planificable. Por tanto, se define un invariante, identificado ***i_4_5_a***, cuya formulación OCL es mostrada por el Código 2.5.

```
context Step inv i_4_5_a:
   Step_Operation.oclIsKindOf(Code_Operation) and
   Step_Schedulable_Resource.oclIsTypeOf(Thread)
   or
   Step_Operation.oclIsKindOf(Message_Operation) and
   Step_Schedulable_Resource.oclIsTypeOf(Communication_Channel)
```

Código 2.5 – Formulación OCL del invariante *i_4_5_a*

b. Existe una pareja de referencias idénticamente tipadas (no necesariamente pertenecientes a la misma clase) y ha de cumplirse una determinada relación de contención entre las poblaciones de elementos de modelo apuntados (una población contenida en la otra o ser disjuntas). Sin embargo, nada en el metamodelo fuerza que en los modelos MAST-2 tales poblaciones de elementos respeten la relación de contención en cuestión.

   Ejemplo: El metamodelo MAST-2 define la clase `Regular_Processor` (subclase de *`Processing_Resource`*) que formula hacia la clase *Timer* las referencias `Timer_List` y `System_Timer`, tal y como se muestra en la Figura 2.11. Mediante la primera se especifica la lista de temporizadores asociados a un procesador, mientras que la segunda apunta al considerado principal de la lista anterior.

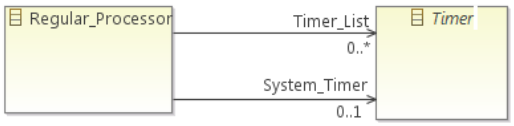


Figura 2.11 – Relación entre las clases `Regular_Processor` y *`Timer`*

   Evidentemente, el temporizador de sistema ha de ser uno de los apuntados por `Timer_List`.

   Sin embargo, nada en el metamodelo asegura que las instancias de `Regular_Processor` presentes en modelos MAST-2 referencien como temporizador de sistema a uno ya incluido en la lista de temporizadores asociados. Por tanto, se define un invariante, identificado ***i_4_1_a***, cuya formulación OCL es mostrada por el Código 2.6 .

```
context Regular_Processor inv i_4_1_a:
   if not System_Timer.oclIsUndefined() then
     Timer_List -> includes(System_Timer)
   else
     true
   endif
```

Código 2.6 – Formulación OCL del invariante *i_4_1_a*

Un estudio en profundidad de todos los tipos posibles de incoherencias que pueden darse en modelos conformes a metamodelos formulados laxos escapa del ámbito de esta Tesis. En la literatura relacionada pueden encontrarse trabajos en tal área, como por ejemplo [102, 103].

2.4 Formalización Amigable de la Información para el Experto de Dominio

Un factor esencial en el diseño de un entorno es hacerlo amigable a los diversos tipos de agentes que lo utilizan. En el contexto de esta sección, el objetivo es facilitar su uso al ingeniero experto de un dominio representado en el entorno, aunque no sea experto en tecnologías MDSE o

incluso ni siquiera en programación. Por ejemplo, el caso del diseño de SDTREs puede considerarse como un macrodominio que engloba campos más especializados (en ocasiones compartidos con otros ámbitos), como el diseño arquitectural y de concurrencia, la especificación de requisitos funcionales y no funcionales, el análisis de planificabilidad, la configuración y despliegue en plataformas de ejecución, etc. Así, los expertos de dominio pueden ser agentes con conocimiento experto únicamente en uno o varios subdominios y no en todas las facetas del desarrollo de SDTREs. Para optimizar su productividad, el entorno debe ofrecer al especialista un nivel de abstracción que corresponda a su dominio.

2.4.1 El enfoque específico de dominio como forma de acercar la información al experto

En general, para trabajar eficientemente al nivel de abstracción de un dominio hay que formular la información en términos de los conceptos en él existentes. De esta manera, la información se presenta al experto de dominio de forma que le resulta familiar, tanto expresiva como semánticamente, pudiendo llevar a cabo su tarea de especificación o diseño (modelado) empleando un juego de primitivas natural en cuanto a su conocimiento experto. Así, el experto no necesita establecer correspondencias entre conceptos de dominio y sus formalizaciones en el entorno de desarrollo, correspondencias que son fuente habitual de errores. Este paradigma de formulación de la información se conoce como *Desarrollo Específico de Dominio* (DSD), cuya particularización dentro de MDSE es el *Modelado Específico de Dominio* (DSM).

Por ejemplo, la adopción de DSM supone que los diseñadores expertos en cada faceta del desarrollo de SDTREs trabajen con modelos cuyos elementos sean instancias de los conceptos que conforman su correspondiente área de especialización:

- **Los expertos en especificación del comportamiento temporal** manejarán modelos compuestos por planificadores, recursos planificables, requisitos temporales, etc.
- **Los encargados del diseño arquitectural** trabajarán con modelos compuestos por componentes, clases activas, clases protegidas, etc.
- **Los responsables de la especificación de requisitos** manejarán modelos conteniendo objetivos, requisitos funcionales, no funcionales, etc.
- Etc.

Lenguajes de propósito general y lenguajes específicos de dominio

El paradigma DSD vertebra el ámbito general del desarrollo de lenguajes para cubrir los diferentes aspectos de un sistema (lenguajes de programación, de consulta de datos, de descripción arquitectural, de especificación de requisitos, etc.). Así, se distingue entre *lenguajes de propósito general* (GPL) que pueden ser aplicados en cualquier dominio o contexto y *lenguajes específicos de dominio* (DSL) que se ajustan a los conceptos de un cierto dominio de trabajo y, por tanto, facilitan las tareas en tal dominio o contexto [20].

Esta clasificación es intuitiva en cualquier dominio, aunque su distinción se establece por convenciones del contexto. Así, en el contexto MDSE, se habla de lenguajes de modelado específicos de dominio (DSML) en contraposición a los de propósito general (GPML), como por

ejemplo UML. De hecho, la característica definitoria del DSM es que los lenguajes de modelado empleados son específicos de dominio.

El empleo de DSLs no es algo nuevo, sino que se ha producido desde los albores de la Computación. Sin embargo, precisamente debido a la aparición y progresivo asentamiento de MDSE y su enfoque DSM, ha sido en las dos últimas décadas cuando ha emergido un renovado interés por este tipo de lenguajes. Así, actualmente es común identificar DSL con DSML, pues es en el ámbito del Modelado donde más lenguajes especializados han surgido y continúan surgiendo. El campo del desarrollo de SDTREs también ha seguido esta tendencia y en la mayor parte de sus áreas concretas puede encontrarse aplicación de DSMLs orientados a ellas, exhibiendo un vocabulario basado en los conceptos propios del dominio en cuestión. Por ejemplo:

- **RDAL** (*Requirements Definition and Analysis Language*), para especificación de requisitos [142].
- **AADL** (*Architecture Analysis and Design Language*), para modelado arquitectural y de concurrencia [77].

Ventajas de adoptar DSM en los entornos

Adoptar un enfoque específico de dominio y emplear DSMLs aporta aún más ventajas a los entornos basados en MDSE. Por ejemplo:

- **Mayor productividad**. Elevar el nivel de abstracción generalmente conduce a incrementar la productividad, no sólo en términos de fase de desarrollo (menos tiempo y recursos necesitados para obtener el producto), sino también en cuanto al mantenimiento.
- **Aplicaciones de mejor calidad**. Los DSMLs incluyen reglas de corrección relativas al dominio, haciendo difícil y a menudo imposible crear especificaciones ilegales o no deseadas. Así, la ocurrencia de errores se previene durante la especificación, que es cuando su corrección es menos costosa.
- **Aumenta el grupo de potenciales desarrolladores**. Ahora los propios expertos de dominio pueden desarrollar por sí mismos aplicaciones, eliminando la necesidad de asistencia de desarrolladores más técnicos, con el consiguiente impulso directo a la productividad.
- **Reducción del tiempo de formación**. De manera natural, el DSM facilita la incorporación de nuevos desarrolladores y ayuda a aquellos menos experimentados a desarrollar aplicaciones con efectividad. Si éstos únicamente necesitan desenvolverse en el dominio, se vuelven productivos más rápidamente que aquellos que también necesiten aprender los dominios de diseño y/o implementación y los mapeos entre dominios.
- **Soporte a la diversidad de vistas**. En el DSM esto se maneja creando diversos DSMLs.
- **Aumento en el número de usuarios**. En el DSM, las especificaciones no son usadas sólo por sus creadores. Los modelos expresados en términos del dominio son fácilmente entendibles por otros participantes, pudiendo esperarse tanto de gestores como de clientes que los examinen, puesto que están basados en conceptos conocidos. Esto redunda en una mejor comunicación y participación.

El lenguaje de modelado MAST-2

Como ejemplo interno al Grupo ISTR, se ha desarrollado el lenguaje MAST-2, orientado a tareas de modelado del comportamiento temporal en el área del análisis de planificabilidad. Para su formalización se ha seguido la visión estructural más extendida actualmente acerca de la constitución de un lenguaje de modelado. Esta visión se expone a continuación.

2.4.2 Elementos para la formalización de los DSMLs

La visión más comúnmente adoptada actualmente considera que un DSML se compone de tres constituyentes básicos esenciales:

- **Sintaxis abstracta (AS)**: Es la definición de los conceptos de modelado así como de sus propiedades e interrelaciones. Constituye la descripción de la estructura del lenguaje y de la forma en que las diferentes primitivas de modelado pueden combinarse, independientemente de cualquier representación o codificación particular.

  Existen varias técnicas para formalizar la AS. Por ejemplo, la más empleada tradicionalmente ha sido definir una gramática en base a una metagramática como EBNF (*Extended Backus-Naur Form*) [104], si bien esto se circunscribe al ámbito de lenguajes textuales. En la actualidad, la técnica más extendida es el metamodelado, posibilitando tanto el enfoque de lenguajes textuales como gráficos. El metamodelo juega un papel análogo al de la gramática; así como una gramática define todas las sentencias válidas, un metamodelo define todos los modelos válidos expresables mediante un lenguaje de modelado.

- **Notación o sintaxis concreta (CS)**: Es el formalismo visual (gráfico o textual) usado para representar los elementos de modelo en editores, constituyendo la referencia para un diseñador en sus actividades de creación o interpretación de modelos. La separación entre AS y CS es una diferencia fundamental respecto a los enfoques tradicionales en ingeniería de lenguajes textuales, donde ambas sintaxis se definen conjuntamente por medio de una gramática EBNF. Esta separación posibilita la definición de más de una notación para un mismo lenguaje de modelado.

  Existen dos tipos básicos de notación:

  - **Sintaxis concreta gráfica** (GCS). La información se representa usando elementos gráficos (flechas, figuras, etc.) formando diagramas.
  - **Sintaxis concreta textual** (TCS). La información se representa mediante texto plano, como en los lenguajes de programación.

  Decantarse por desarrollar un DSML dotándolo de notación gráfica o de notación textual no es una decisión trivial. Ambos enfoques presentan beneficios e inconvenientes y en ambos casos existe diversidad de *frameworks* para construcción de DSMLs. En sistemas sencillos, la notación gráfica facilita la capacidad de comprensión global que posee el operador humano. Sin embargo, en sistemas más complejos esta capacidad humana se pierde y es más sencillo y potente utilizar la notación textual. Afortunadamente, puesto que la separación entre AS y CS permite dotar a un DSML de más de una notación, los enfoques gráfico y textual no son mutuamente excluyentes. En esta Tesis se utiliza sólo el enfoque textual [105].

- **Semántica**: Es la descripción del significado detallado de sus elementos conceptuales, posibilitando su uso correcto. A pesar de que no tiene sentido desarrollar un lenguaje sin especificar su semántica, ésta es a menudo descuidada, entre otras cosas porque no es algo trivial. De hecho es más propio del experto de dominio que del experto en MDSE. En la Tesis no se han abordado las estrategias que existen para formalizarla: denotacional, translacional, operacional, etc.

2.4.3 Herramientas para la creación de DSMLs textuales.

La creación de un DSML textual incluye tanto la especificación del lenguaje como la construcción del correspondiente *language workbench* [106-108], esto es, el conjunto de herramientas de soporte (editores, *parsers*, etc.). Para su desarrollo es necesario disponer de una plataforma al efecto. Algunas alternativas son las siguientes:

- Proyectos Eclipse:
  - ***Xtext*** [37] [38] [109]: Pertenece al subproyecto TMF (*Textual Modeling Framework*) dentro de EMP, y ha sido desarrollado por Itemis AG[39] (Kiel).
  - **TCS**[40] (*Textual Concrete Syntax*) [110]: También pertenece a TMF y ha sido desarrollado por Frederic Jouault (Escuela de Minas de Nantes) y ATLAN-Mod[41].
  - **IMP**[42] (*IDE Meta-Tooling Platform*) [111]: Forma parte de Eclipse, aunque no pertenece a TMF y ha sido desarrollado IBM Watson Research[43].
- Basadas en Eclipse sin ser proyecto Eclipse oficial:
  - **TEF**[44] (*Textual Editing Framework*) [112]: Desarrollado por Markus Scheidgen en su PhD en la TU Berlín.
  - **EMFText**[45] [113]: Forma parte del proyecto DropsBox[46] (*Dresden Open Software Toolbox*), y ha sido desarrollado por DevBoost y el Grupo de Tecnología Software en la TU Dresden.
  - **Monticore**[47] [114]: Desarrollado por el Grupo de Ingeniería de Software de la TU Braunschweig[48].
- No relacionadas con Eclipse:
  - **MPS** [49] (Meta-Programming System) [115]: Desarrollado por la empresa *JetBrains*.

[37] http://projects.eclipse.org/projects/modeling.tmf.xtext
[38] http://www.eclipse.org/Xtext
[39] http://www.itemis.com
[40] http://www.eclipse.org/gmt/tcs
[41] http://www.emn.fr/z-info/atlanmod/index.php/Main_Page
[42] http://projects.eclipse.org/projects/technology.imp
[43] http://www.research.ibm.com/labs/watson
[44] http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/index.html
[45] http://www.emftext.org/index.php/EMFText
[46] http://www.dropsbox.org/index.php/DropsBox
[47] http://www.monticore.de
[48] http://www.se-rwth.de
[49] https://www.jetbrains.com/mps

En esta Tesis se ha utilizado exclusivamente *Xtext*, y se le dedica la siguiente subsección. La razón de su elección es que al ser un proyecto oficial de Eclipse es el que ofrece mayor cobertura en términos de comunidad de usuarios, documentación disponible, soporte, actividad y evolución.

2.4.4 Xtext

Especificación de una TCS en Xtext

En Xtext, la notación textual para un DSML se formaliza en forma de gramática expresada en términos del *Xtext Grammar Language*. Xtext permite tomar como punto de partida la AS del DSML formalizada como modelo Ecore para generar a partir suyo una gramática por defecto, completa y funcional, que puede ser refinada a continuación. Por otro lado, una gramática también puede ser escrita desde cero y Xtext permite inferir a partir suyo el correspondiente metamodelo. Esta segunda opción se antoja menos moderna, pues la descripción de los conceptos del dominio queda englobada en el propio desarrollo de la gramática. Por ello, en esta Tesis se ha apostado por la primera vía.

En cualquier caso, una gramática Xtext, independientemente de si se ha desarrollado manualmente o ha sido generada, está constituida por reglas gramaticales, distinguiéndose:

- **Reglas terminales**. Describen los patrones léxicos que ha de saber reconocer el *parser* cuando analice los modelos. Por tanto, en una gramática Xtext, el conjunto de reglas terminales es el bloque básico.
- **Reglas correspondientes a los tipos primitivos** (`EDataType`) definidos en el metamodelo. Su signatura es mostrada por el Código 2.7:

```
Nombre_Regla returns Nombre_EDataType:
   Especificación del patrón léxico asociado, típicamente
   mediante una combinación de reglas terminales;
```

Código 2.7 – Signatura de regla correspondiente a un tipo primitivo

- **Reglas correspondientes a los tipos enumerados** (`EEnumType`) definidos en el metamodelo. Su signatura es mostrada por el Código 2.8:

```
Nombre_Regla returns Nombre_EEnumType:
   Especificación de los literales que componen el EEnumType;
```

Código 2.8 – Signatura de regla correspondiente a un tipo enumerado

- **Reglas correspondientes a las clases** (`EClass`) definidas en el metamodelo. Su signatura para el caso de una clase abstracta es mostrada por el Código 2.9:

```
Nombre_Regla returns Nombre_EClass:
   Nombre_SubclaseConcreta_1 |
   ...
   Nombre_SubclaseConcreta_N;
```

Código 2.9 – Signatura de regla correspondiente a clase abstracta

Y para una clase concreta es mostrada por el Código 2.10:

```
Nombre_Regla returns Nombre_EClass:
  'Nombre_EClass' Nombre_EAttr_RolID=Nombre_EData_Type '{'

    // EAttr_1 -> multiplicidad [1]
    'Nombre_EAttr_1' Nombre_EAttr_1=Nombre_EData_Type

    // EAttr_2 -> multiplicidad [0..1]
    ('Nombre_EAttr_2' Nombre_EAttr_2=Nombre_EData_Type)?

    // EAttr_3 -> multiplicidad [1..*]
    'Nombre_EAttr_3('Nombre_EAttr_3+=[Nombre_EData_Type](","
                    Nombre_EAttr_3+=[Nombre_EData_Type])*')'

    // EAttr_4 -> multiplicidad [0..*]
    ('Nombre_EAttr_4('Nombre_EAttr_4+=[Nombre_EData_Type](","
                     Nombre_EAttr_4+=[Nombre_EData_Type])*')')?

    // ERef_1 -> multiplicidad [1]
    'Nombre_ERef_1'Nombre_ERef_1=[Nombre_EClass|EString]

    // ERef_2 -> multiplicidad [0..1]
    ('Nombre_ERef_2'Nombre_ERef_2=[Nombre_EClass|EString])?

    // ERef_3 -> multiplicidad [1..*]
    'Nombre_ERef_3('Nombre_ERef_3+=[Nombre_EClass|EString](","
                   Nombre_ERef_3+=[Nombre_EClass|EString])*')'

    // ERef_4 -> multiplicidad [0..*]
    ('Nombre_ERef_4('Nombre_ERef_4+=[Nombre_EClass|EString](","
                    Nombre_ERef_4+=[Nombre_EClass|EString])*')')?

    // ERef_5 -> multiplicidad [1], con contención
    Nombre_ERef_5=Nombre_EClass

    // ERef_6 -> multiplicidad [0..1], con contención
    (Nombre_ERef_6=Nombre_EClass)?

    // ERef_7 -> multiplicidad [1..*], con contención
    Nombre_ERef_7+=Nombre_EClass(","Nombre_ERef_7+=Nombre_EClass)*

    // ERef_8 -> multiplicidad [0..*], con contención
    (Nombre_ERef_8+=Nombre_EClass(","Nombre_ERef_8+=Nombre_EClass)*)?

'}';
```

Código 2.10 – Signatura de regla correspondiente a clase concreta

Cuando cualquiera de entre los patrones de texto definidos por este último tipo de regla es identificado en el texto del modelo, se crea una instancia de la clase correspondiente, y asimismo, cuando un modelo contiene una instancia de una clase, se escribe en el texto del modelo el correspondiente patrón.

La subsección 2.4.5 se apoya en el lenguaje de modelado MAST-2 para ilustrar la especificación de una gramática Xtext.

Provisión de un editor

A partir de la especificación de una gramática, Xtext es capaz de generar un sofisticado editor que posibilita la visualización enriquecida de la información de los modelos así como su edición asistida. Las características por defecto de tales editores son:

- **Una configuración tipográfica** consistente en la consideración de seis categorías léxicas básicas (palabras clave[50], comentarios, números, *strings*, puntuaciones y símbolos inválidos), definiendo un estilo tipográfico diferente para cada una de ellas. Éstos están todos basados

[50] Nombres de clases, de atributos, de referencias, etc.

en la misma fuente pero diferenciados por características como la coloración o el resaltado en negrita. Además, se define un séptimo estilo tipográfico en la misma línea de los anteriores para todo *token* no perteneciente a alguna de esas seis categorías léxicas básicas.

- **Disponibilidad de un asistente de contenido** que ofrece completado relativo a:
  - Palabras clave y delimitadores.
  - Atributos de tipo enumerado.
  - La asignación de referencias entre objetos.

  En los dos últimos casos, la conducta por defecto del asistente se basa exclusivamente en el metamodelo de dominio, haciendo caso omiso a cualquier restricción especificada sobre él.
- ***Live validation,*** señalando las faltas de conformidad entre modelo y gramática.
- **Importación/exportación a ficheros XML/XMI**.

Una vez generada toda la infraestructura del editor puede procederse a la personalización de muchos de sus aspectos, siempre mediante modificación directa del código Java generado como parte suya. Por ejemplo, es posible:

- **Definir una configuración tipográfica personalizada**, tanto simplemente modificando los estilos de las categorías léxicas predefinidas como definiendo nuevas categorías con sus respectivos estilos.
- **Refinar el asistente de contenido** para que, a la hora de ofrecer propuestas de completado, contemple las posibles restricciones especificadas sobre el metamodelo.

Se ha realizado el estudio del diseño de una metaherramienta que, en base a modelos que describan las extensiones con que se desea enriquecer un editor Xtext respecto a su funcionalidad por defecto, modifique el código del editor para que tenga los comportamientos buscados. Se han explorado dos estrategias:

a. **Procesamiento del modelo mediante un código genérico**, independiente del metamodelo de dominio, y por tanto aplicable para cualquier editor. Éste va a ser el caso de la definición de una configuración tipográfica personalizada.
b. **Procesamiento del modelo mediante una herramienta M2T** que posibilite la generación automática del código que implementa la extensión para un editor concreto. Éste va a ser el caso del refinamiento del asistente de contenido para que contemple las restricciones especificadas sobre las referencias existentes en el metamodelo de dominio.

En el momento de escribir esta memoria, esta metaherramienta no está aún disponible.

2.4.5 Gramática y editor para el lenguaje MAST-2

Gramática

Para el lenguaje de modelado MAST-2, cuya AS se encuentra formalizada a través del metamodelo de igual nombre, formulado mediante el lenguaje de metamodelado Ecore (**`Mast2.ecore`**), se ha desarrollado una TCS formalizada a través de una gramática formulada mediante el *Xtext Grammar Language* (**`Mast2.xtext`**). Ésta ha sido generada a partir del metamodelo, procediendo posteriormente a su refinado y personalización.

La Figura 2.12 muestra la cabecera de la gramática `Mast2.xtext`.

```
Mast2.xtext
1  grammar es.unican.ctr.xtext.Mast2 with org.eclipse.xtext.common.Terminals
2
3  import "http://mast.unican.es/ecoremast/Mast2"
4  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
```

Figura 2.12 – Cabecera de la gramática `Mast2.xtext`

`Mast2.xtext` incluye las reglas terminales mostradas en el Código 2.11:

```
/*
 * Lexical patterns for strings
 */
terminal ID:
   ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'.')*;

/*
 * Lexical patterns for integer numbers
 */
terminal POSITIVE_INTEGER:
   ('1'..'9')('0'..'9')*;

terminal NON_NEGATIVE_INTEGER:
   ('0'..'9')('0'..'9')*;

/*
 * Lexical patterns for floating point numbers
 */
terminal POSITIVE_FLOATING:
   NON_NEGATIVE_INTEGER '.' '0'* POSITIVE_INTEGER*;

/*
 * Lexical patterns for floating point numbers in Scientific Notation
 */
terminal POSITIVE_FLOATING_SN:
   NON_NEGATIVE_INTEGER '.' '0'* POSITIVE_INTEGER* 'E' '-'? POSITIVE_INTEGER;
```

Código 2.11 – Reglas terminales en `Mast2.xtext`

A continuación se muestran algunas estructuras del metamodelo MAST-2 – tipos primitivos, tipos enumerados y (conjuntos de) clases – junto a su correspondiente expresión en forma de reglas gramaticales. En el caso de clases, un fragmento de un modelo MAST-2 expresado textualmente acompaña a efectos ilustrativos.

- **Tipos primitivos**.
  La Figura 2.13 muestra los tipos primitivos (`EDataType`) definidos en `Mast2.ecore`.

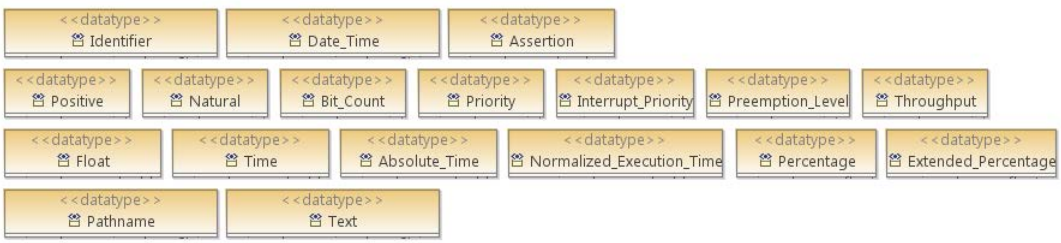


Figura 2.13 – Tipos primitivos en `Mast2.ecore`

A cada uno le corresponde una *EDataType rule* de entre las mostradas por el Código 2.12:

```
Identifier returns Identifier:
   ID;
Date_Time returns Date_Time:
   ID;
Assertion returns Assertion:
   'true' | 'false';
Positive returns Positive:
   POSITIVE_INTEGER;
Natural returns Natural:
   POSITIVE_INTEGER | '0';
Bit_Count returns Bit_Count:
   POSITIVE_INTEGER;
Priority returns Priority:
   POSITIVE_INTEGER;
Interrupt_Priority returns Interrupt_Priority:
   POSITIVE_INTEGER;
Preemption_Level returns Preemption_Level:
   POSITIVE_INTEGER;
Throughput returns Throughput:
   INT;
Float returns Float:
   POSITIVE_FLOATING;
Time returns Time:
   POSITIVE_FLOATING | POSITIVE_FLOATING_SN;
Absolute_Time returns Absolute_Time:
   POSITIVE_FLOATING | POSITIVE_FLOATING_SN;
Normalized_Execution_Time returns Normalized_Execution_Time:
   POSITIVE_FLOATING | POSITIVE_FLOATING_SN;
Percentage returns Percentage:
   POSITIVE_FLOATING | POSITIVE_FLOATING_SN;

EString returns ecore::EString:
   ID;
```

Código 2.12 – Reglas EDataType en `Mast2.xtext`

- **Tipos enumerados**.

  La Figura 2.14 muestra los tipos enumerados (`EEnumType`) definidos en `Mast2.ecore`.

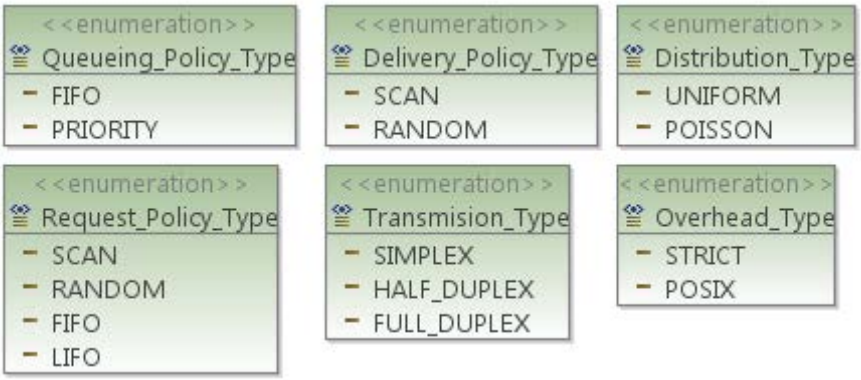


Figura 2.14 – Tipos enumerados en `Mast2.ecore`

A cada uno le corresponde una *EEnumType rule* de entre las mostradas en el Código 2.13:

```
enum Overhead_Type returns Overhead_Type:
   POSIX | STRICT;
enum Queueing_Policy_Type returns Queueing_Policy_Type:
   FIFO | PRIORITY;
enum Transmision_Type returns Transmision_Type:
   SIMPLEX | HALF_DUPLEX | FULL_DUPLEX;
enum Distribution_Type returns Distribution_Type:
   UNIFORM | POISSON;
enum Delivery_Policy_Type returns Delivery_Policy_Type:
   SCAN | RANDOM;
enum Request_Policy_Type returns Request_Policy_Type:
   FIFO | LIFO | RANDOM | SCAN;
```

Código 2.13 – Reglas EEnumType en `Mast2.xtext`

- **(Conjuntos de) Clases**.

  La Figura 2.15 se centra en las clases `Mast_Model` y *`Model_Element`*. A continuación, el Código 2.14 y el Código 2.15 muestran las reglas correspondientes.

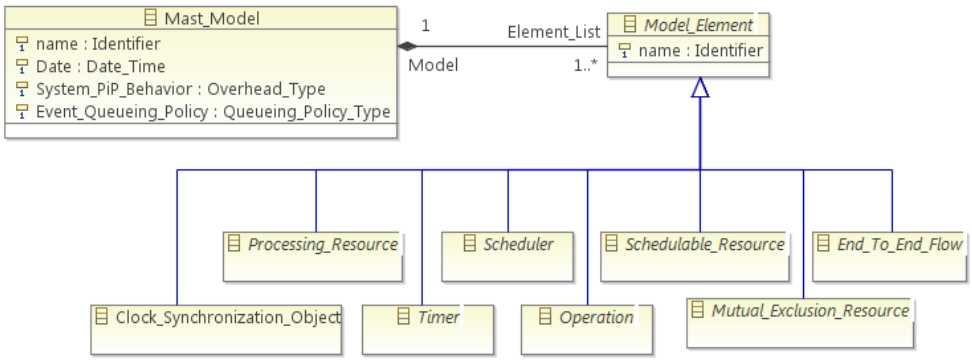


Figura 2.15 – Clases principales en `Mast2.ecore`

```
Model_Element returns Model_Element:
   Clock_Synchronization_Object | Ticker | Alarm_Clock |
   Regular_Processor |
   Packet_Based_Network_Impl | RTEP_Network | AFDX_Link |
   Regular_Switch_Impl | AFDX_Switch | Regular_Router |
   Primary_Scheduler | Secondary_Scheduler |
   Thread_Impl | Communication_Channel_Impl |
   Virtual_Schedulable_Resource | Virtual_Communication_Channel |
   Immediate_Ceiling_Mutex | Priority_Inheritance_Mutex | SRP_Mutex |
   Simple_Operation | Composite_Operation_Impl | Enclosing_Operation |
   Message | Composite_Message |
   Regular_End_To_End_Flow;
```

Código 2.14 – Regla `Model_Element`

```
Mast_Model returns Mast_Model:
   'Mast_Model' name=Identifier '{'
      (
      'Date' Date=Date_Time &
      'System_PiP_Behavior' System_PiP_Behavior=Overhead_Type &
      'Event_Queueing_Policy' Event_Queueing_Policy=Queueing_Policy_Type
      )
      Element_List+=Model_Element ( "," Element_List+=Model_Element)*
   '}';
```

Código 2.15 – Regla `Mast_Model`

La Figura 2.16 se centra en la clase *Timer* y sus subclases `Ticker` y `Alarm_Clock`. A continuación, el Código 2.16 y el Código 2.17 muestran las reglas correspondientes.

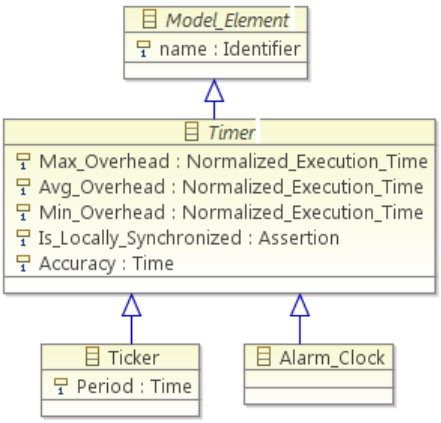


Figura 2.16 – Clase Timer y sus subclases

```
Timer returns Timer:
   Alarm_Clock | Ticker;
```

Código 2.16 – Regla `Timer`

```
Ticker returns Ticker:
   'Ticker' name=Identifier '{'
      (
      'Max_Overhead' Max_Overhead=Normalized_Execution_Time &
      'Avg_Overhead' Avg_Overhead=Normalized_Execution_Time &
      'Min_Overhead' Min_Overhead=Normalized_Execution_Time &
      'Is_Locally_Synchronized' Is_Locally_Synchronized=Assertion &
      'Accuracy' Accuracy=Time &
      'Period' Period=Time
      )
   }';
```

Código 2.17 – Regla `Ticker`

Editor

A partir de la especificación de `Mast2.xtext` se ha empleado la capacidad de Xtext para generación de un editor textual que posibilite la visualización enriquecida de la información de los modelos MAST-2 así como su edición asistida. Posteriormente se ha procedido a la personalización de su configuración tipográfica y al refinamiento del asistente de contenido.

- **Configuración tipográfica**. Se han mantenido inalterados los estilos tipográficos correspondientes a las categorías léxicas básicas pero se han definido tres nuevas categorías: nombre de clase, nombre de atributo y nombre de referencia. Para cada una se ha definido un estilo propio.
- **Asistente de contenido**. Se ha mejorado su funcionalidad para que las propuestas de completado que ofrezca al ser invocado en la asignación de referencias entre objetos contemplen las *Restricciones MAST-2*.

2.5 Persistencia e Intercambio de Modelos

2.5.1 Necesidad de persistencia de los modelos

Los modelos manejados en un entorno pueden ser *efímeros*, esto es, su ciclo de vida se desarrolla en el seno de la ejecución de la herramienta o aplicación que los crea, o pueden ser *persistentes* dentro de un repositorio, bien porque son parte del entorno o bien porque, una vez generados, su información va a ser utilizada en posteriores sesiones de trabajo. Esta distinción es un aspecto esencial en los entornos, debiendo dar soporte a ambos *estados de existencia* de los modelos.

Modelos en forma activa

Los modelos categorizados como efímeros residen en el espacio de memoria de las aplicaciones (modelos en forma activa) y es allí donde se gestiona la información que aportan. En este caso, los modelos desaparecen al finalizar la ejecución de la aplicación que los ha creado (o como muy tarde al terminar la sesión de trabajo del entorno).

Modelos persistidos

Los modelos persistidos se encuentran almacenados en un repositorio sobre un medio de almacenamiento persistente o no volátil (disco duro, memoria *flash*, etc.). Para ello se ha de representar su información con un formato compatible con el medio que se utiliza:

- **Ficheros**. La información del modelo se almacena como una secuencia de bytes en el medio físico persistente.
- **BBDD**. La información se almacena en un servidor persistente, local o remoto, utilizando diferentes paradigmas: BBDD basadas en clave-valor, relacionales, orientadas a objetos, etc.

En esta Tesis se considera sólo la persistencia de los modelos en forma de ficheros textuales. Por ello, los mecanismos de persistencia están íntimamente relacionados con la capacidad y forma de secuenciar la información de los modelos.

2.5.2 Comunicación con otros entornos o herramientas externas

Los entornos de desarrollo pueden y suelen ser complejos e involucrar la actuación de herramientas externas, bien *standalone* o integradas en otros entornos. Cuando las tecnologías base del entorno y de la herramienta externa no son compatibles para interpretar los modelos en una única forma activa, se necesita una estrategia de intercambio basada en una formalización estándar que ambas reconozcan. La solución más generalizada es el intercambio a través de flujos (*stream*) de la información de los modelos serializada en caracteres textuales codificados de acuerdo con un estándar y soportados a través de algún recurso proporcionado por los sistemas operativos o las redes de comunicación.

En la siguiente subsección se presenta la serialización de los modelos como solución consolidada, y por tanto adoptada en esta Tesis, para habilitar que los modelos rebasen los límites de un entorno, posibilitando tanto su persistencia como su transferencia.

2.5.3 Serialización de modelos

Se entiende por serialización de modelos el *proceso de traducir un modelo a una forma binaria o textual que permita su almacenamiento persistente o su transmisión a través de una red y su posterior reconstrucción, bien en el mismo entorno o en otro distinto*. Así, la serie de bytes resultante puede usarse para crear un clon semánticamente idéntico al modelo original. La operación opuesta, extraer un modelo a partir de una serie de bytes se conoce como deserialización.

Formatos de serialización

Existen formatos de serialización binaria, diseñados para representar eficientemente la información aunque no sea fácilmente legible por los operadores humanos. De forma alternativa, existen formatos de serialización textual, que codifican la información de forma menos eficiente pero que pueden ser más fácilmente interpretados por un operador humano. Desde finales de los 90 se ha generalizado el uso de las alternativas textuales, principalmente de la mano del formato XML, ya que al estar asociado a las tecnologías WEB, está soportado por todas las plataformas, con independencia de cuál es su tecnología o su naturaleza.

2.5.4 El estándar XMI como formato (de serialización textual de modelos) dominante.

La generalización de XML como formato de serialización textual de datos ha requerido completarlo con unos criterios de estandarización complementarios, bajo el nombre XMI (*XML Metadata Interchange*), para facilitar la intercolaboración entre entornos y herramientas MDSE.

XMI es un estándar de OMG cuya primera especificación data de 1998, poco después de que fuese finalizado XML 1.0 y cuya versión actual desde 6-Abril-2014 es XMI v2.4.2[51]. Su función es conectar modelado con XML, definiendo una forma simple de serializar modelos en formato XML, con la consiguiente posibilidad de persistirlos como documentos XML (documentos XMI). La estructura de un documento XMI concuerda con la del correspondiente modelo, con los mismos nombres y la misma jerarquía de elementos que sigue la jerarquía de contención del modelo. Como resultado, la relación entre un modelo y su serialización XMI es fácil de entender.

Básicamente, la especificación XMI define los siguientes aspectos de la formulación de un documento XML:

- La representación XML de la información se realiza en base a elementos y atributos.
- Establece los mecanismos para referenciar objetos dentro del mismo u otros documentos.
- La validación de los documentos XMI utiliza W3C-Schemas.
- Permite definir identificadores de los objetos, lo cual permite que los objetos sean referenciados en base a IDs y a UUIDs estándar.

2.5.5 Persistencia en EMF

Uno de los puntos fuertes de EMF es la posibilidad de persistir objetos y de poder referenciar a otros objetos persistidos, pues esto constituye la base para una fina integración de datos entre

[51] http://www.omg.org/spec/XMI/2.4.2

aplicaciones. Para gestionar la persistencia de objetos (y, por extensión, de modelos) EMF define un *framework* con recursos para almacenar y recuperar modelos. Este *framework de persistencia* viene implementado por el API de soporte incluido en EMF.

EMF permite serialización XML altamente configurable, incluyendo soporte al estándar XMI 2.0, el cual, de hecho, se toma como formato de representación canónico. Asimismo, permite serializar modelos Ecore (metamodelos) como EMOF, facilitando la interoperabilidad entre herramientas de modelado. Sin embargo, EMF no requiere que la persistencia esté basada en XML, ni siquiera basada en *stream*. El API de persistencia proporcionado es suficientemente flexible para soportar cualquier tipo de almacenamiento e incluso para persistir objetos de diferentes maneras pudiendo mantener referencias entre ellos.

Framework de persistencia

El *framework* de persistencia de EMF está constituido básicamente por los siguientes elementos:

- **Recurso**. Unidad básica de persistencia en EMF, asociada a una localización de almacenamiento físico (por ejemplo, típicamente un fichero). Representa un contenedor para objetos que han de ser persistidos conjuntamente, junto a sus correspondientes objetos contenidos. Por tanto, un recurso incluye por defecto todo el árbol de objetos contenido por cada uno de los objetos de su lista de contenidos. Típicamente, aunque no necesariamente, un recurso se corresponde con un modelo.

  El *tipo de un recurso* se corresponde básicamente con la forma persistente de un modelo a ser producida y consumida.

- **Factoría de recursos.** Define una manera de crear recursos. Una factoría simplemente sabe cómo crear recursos de un cierto tipo. La factoría XMI es la factoría por defecto en EMF.
- **Registro de factorías**. Posibilita seleccionar la factoría apropiada para crear recursos, en base a un URI especificado al efecto. Un registro de factorías mantiene dos mapas en los que se registran factorías, los cuales son consultados para obtener la factoría apropiada para un URI dado.
- **Conjunto de recursos (*set*)**. El *framework* define el concepto de conjunto de recursos o *set* con el propósito de permitir la gestión de referencias entre objetos localizados en diferentes recursos. Así, un *set* actúa como contenedor de recursos que son accedidos conjuntamente.
- **URI**. Los URIs desempeñan un papel crucial en el *framework*, siendo usados extensivamente para identificar de forma unívoca y para localizar tanto recursos como objetos dentro de recursos, así como para identificar paquetes.

  Los recursos se identifican unívocamente mediante un URI, y asimismo cada objeto del recurso se identifica unívocamente por un *fragmento URI*, con lo que se facilita la navegación directa entre objetos de diferentes recursos sin necesidad de recorrer los arboles de objetos de los recursos.

2.6 Clasificación y Organización de la Información en un Entorno

Ya se ha expuesto anteriormente que la información involucrada en un proceso de desarrollo de SDTREs es masiva, heterogénea y fuertemente interreferenciada. Apostar por MDSE y otros espacios tecnológicos estructuralmente afines para gestionar información tan diversa e

interrelacionada proporciona la capacidad de representarla mediante modelos que se corresponden con fragmentos parciales de información relativos a las distintas fases y aspectos del ciclo de desarrollo. Se trata de un enfoque diametralmente opuesto al de las herramientas CASE tradicionales, donde se aboga por artefactos de información globales y muy extensos (también llamados modelos, aunque no en sentido estricto MDSE), en los que tiene cabida toda la información manejada.

La formulación de la información fragmentándola en conjuntos de modelos junto a la capacidad de ser éstos persistidos en los entornos de desarrollo, implica la conveniencia de un modelo organizativo que permita su explotación ágil y eficiente. En este contexto, se emplea el término ***recurso*** como sinónimo de ***artefacto de información en un entorno***. En principio se distinguen dos tipos de recursos:

- **Recurso MDSE**. Este término cubre cualquier recurso interpretable en términos MDSE, es decir, formulado en base al TS *Modelware* u otro (*Grammarware*, XML, etc.) que permita una interpretación MDSE según la Arquitectura de 3+1 niveles. De ahí que se emplee también genéricamente *modelo*, independientemente de si pertenece al metanivel M1 (modelo en XMI, documento XML ordinario, modelo en texto plano, etc.) o al metanivel M2 (metamodelo, W3C-Schema, gramática, etc.).
- **Recurso No-MDSE**. Ficheros de documentación, imágenes, ficheros de configuración, *scripts*, etc.

Puesto que esta Tesis tiene por objeto los entornos basados en MDSE, lógicamente se considera que la organización y clasificación de la información ha de girar en torno al primer tipo de recursos (modelos), mientras que la ubicación y organización de los segundos irá en función de los primeros.

2.6.1 Criterios de clasificación de los modelos

Se considera el siguiente conjunto de criterios para la clasificación de los modelos contenidos en un entorno durante un proyecto de desarrollo de SDTREs:

- **El propio SDTRE bajo desarrollo**.
- **Fase del ciclo de desarrollo**. Bajo este criterio, los modelos se clasifican atendiendo a la etapa a la que pertenecen dentro del proceso de desarrollo. Así, pueden tenerse modelos de especificación, análisis, diseño, etc.
- **Aspecto del desarrollo**. Cada fase puede desglosarse en distintos aspectos, los cuales definen subcategorías para los modelos. Así, por ejemplo, los modelos correspondientes a la fase de especificación pueden ser modelos de especificación funcional o de especificación no funcional.

  Dentro de un aspecto cabe definir subaspectos o dominios más concretos. Por ejemplo, los modelos de especificación no funcional pueden serlo de comportamiento temporal, relativos al consumo de energía o modelos de QoS, entre otros.
- **Equipo desarrollador**. Diferentes desarrolladores o grupos pueden trabajar simultáneamente en una misma tarea. Los modelos producidos comparten las mismas categorías relativas a la fase o aspecto del desarrollo, pero se diferencian por su origen.

- **Metodología / Herramientas**. Similar al punto anterior, modelos que pertenecen a la misma categoría en relación a la fase o aspecto del desarrollo pueden distinguirse por haber sido construidos o generados aplicando distintas metodologías o empleando diferentes herramientas, por ejemplo cuando el objetivo es evaluar diversas alternativas tecnológicas.
- **Espacio tecnológico**. Los recursos generados y gestionados en los entornos objeto de esta Tesis son esencialmente modelos pero que, dependiendo del TS escogido para su formulación y persistencia, pueden adoptar la forma de documentos XML ordinarios, documentos XMI o documentos de texto plano. A su vez, los artefactos a nivel M2 que formalizan tales modelos son respectivamente W3C-Schemas, metamodelos propiamente dichos o gramáticas.
- **Metanivel**. Perspectiva centrada en clasificar los diversos recursos MDSE en función de la Arquitectura de 3+1 Niveles, que, aunque propia de MDSE, también estructura al resto de los espacios tecnológicos considerados, como quedó expuesto en la sección 2.1. Por tanto, se trata de una visión aplicable a cualquier repositorio de información basado en *Modelware* y demás espacios estructuralmente afines, y por consiguiente aplicable a la noción de entorno considerada en esta Tesis.
- **Versionado**. Los modelos atraviesan diferentes versiones según van siendo actualizados, enriquecidos o refinados.

2.6.2 Soluciones de localización de la información

Como raíz en cuanto a la jerarquía de contención de los recursos se considera la noción de ***espacio de trabajo*** o *workspace*. En el contexto de esta Tesis, un *workspace* representa el contenedor global de toda la información relativa a los diferentes aspectos, vistas y fases en el desarrollo de SDTREs, la cual es manejada a través del marco uniforme que es el entorno.

Un *workspace* típicamente se corresponde con un cierto proyecto de desarrollo de suficiente entidad o envergadura como para ser considerado un escenario cerrado en sí mismo. Así, las sesiones de trabajo mediante un entorno tienen lugar sobre un *workspace* determinado, de manera que las diferentes tareas se llevan a cabo fundamentalmente dentro de su ámbito.

2.6.3 Estrategias de organización de los modelos

Cualquier propuesta de organización de los modelos en un entorno pasa por tomar como base uno o varios de los criterios de clasificación expuestos. El número de criterios que es posible aplicar de forma simultánea depende fundamentalmente de la *estrategia de gestión de recursos* que ofrece la infraestructura MDSE sobre la que se construye el entorno.

Dos estrategias básicas son:

- **Por ubicación física en una estructura de contenedores**. Es el paradigma de los tradicionales sistemas de ficheros. Cada modelo se ubica en un y sólo un contenedor, pudiendo existir anidación de unos contenedores en otros.
- **Mediante etiquetado**. En este caso no es necesaria ninguna estructura de contención, sino que todos los modelos pueden residir en un mismo repositorio común pero con diferentes etiquetas que especifican su clasificación y organización.

Este paradigma puede visualizarse también en forma de contenedores, pero sin olvidar que asignar un modelo a uno de ellos no significa agregación, sino etiquetado. Así, un mismo modelo puede asignarse a más de uno de tales pseudocontenedores, lo cual resultaría imposible bajo el paradigma anterior, dado su carácter mutuamente exclusivo.

Contenedores

Bajo esta estrategia, la manera de implementar la aplicación de los criterios clasificatorios (subsección 2.6.1) es identificando cada nivel de anidación con uno de tales criterios. La Figura 2.17 muestra esta identificación entre criterios y niveles de contención, donde se ha optado por el SDTRE bajo desarrollo como criterio clasificatorio de primer nivel, situando al equipo de desarrollo a continuación y así sucesivamente.

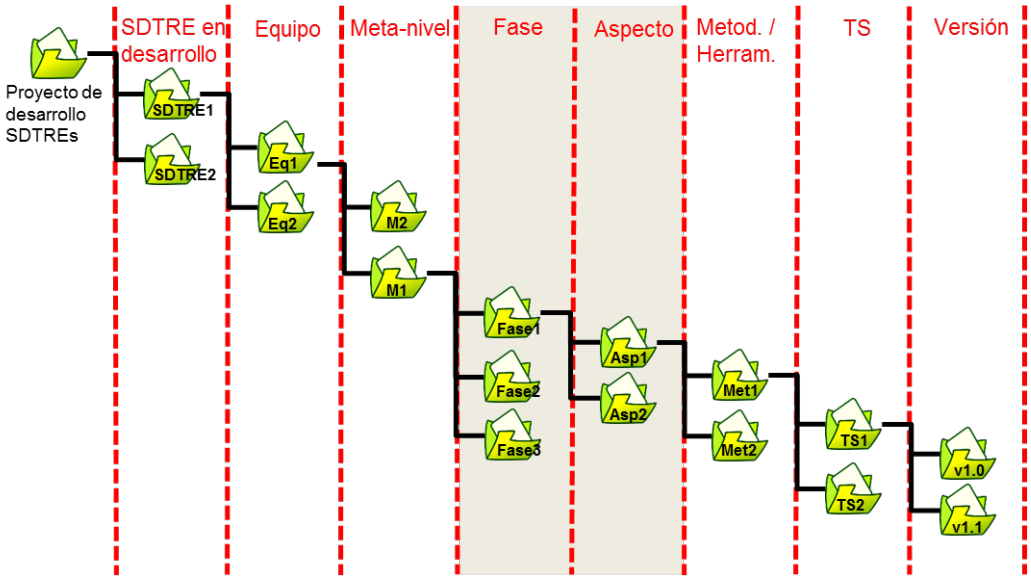


Figura 2.17 – Organización de los modelos aplicando anidación de contenedores

Esta estrategia permite múltiples configuraciones alternativas según el orden en que se desee aplicar los criterios clasificatorios, permutando los niveles de anidación. Por ejemplo, lo mostrado por la Figura 2.17 es simplemente una posible configuración, pues algunos de los niveles de anidamiento son claramente intercambiables, como pueden ser los correspondientes al TS y al versionado.

Sin embargo, no todos los niveles de contención son permutables, como sucede en la Figura 2.17 con los dos centrales (fase y aspecto – sombreados –), pues son criterios entre los que existe una subordinación semántica: los aspectos lo son de una fase del ciclo de desarrollo.

Por último, cabe mencionar cómo la Figura 2.17 permite advertir que el carácter mutuamente exclusivo de la contención implica la necesidad de replicar recursivamente estructuras de contenedores anidados. Así, del equipo de desarrollo *Eq2* pendería una réplica de la estructura completa que pende de *Eq1*.

Etiquetado

La Figura 2.18 muestra una estrategia organizativa basada en el paradigma de etiquetado y alternativa a la expuesta en el apartado anterior. Ejemplifica cómo bajo esta estrategia es posible asignar un modelo a más de una categoría (pseudocontenedor) perteneciente a un

mismo criterio clasificatorio. El modelo mostrado en la Figura 2.18 queda caracterizado como un modelo que describe simultáneamente el comportamiento temporal y el consumo energético del sistema bajo desarrollo "Robot", ha sido desarrollado conjuntamente por dos equipos distintos y representa su versión 1.0.1 serializada como XMI.

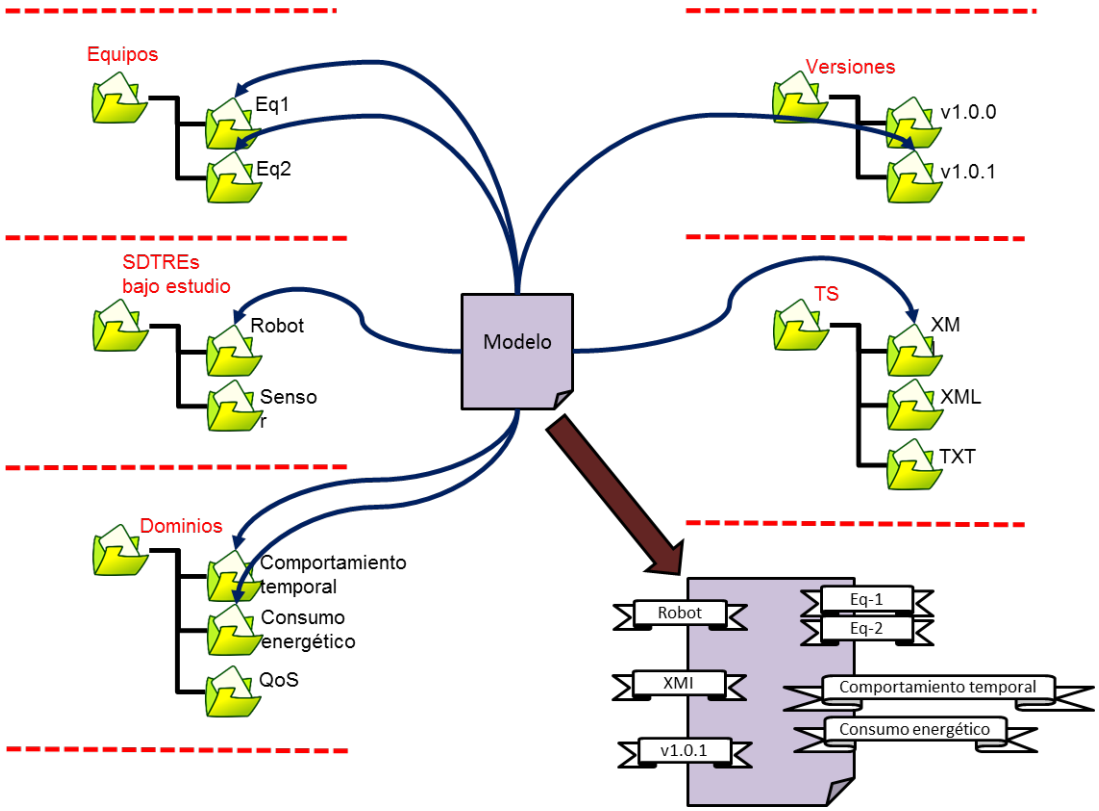


Figura 2.18 – Organización de los modelos aplicando etiquetado

2.6.4 Paradigma de gestión de recursos en Eclipse

En Eclipse, cada sesión de trabajo se lleva a cabo sobre un directorio del sistema de ficheros. Esta ubicación se conoce como *workspace* y es donde reside la información correspondiente a las tareas de desarrollo. Para referirse a los artefactos existentes en un *workspace*, tanto de información como de organización (contención y vinculación) de la propia información, se emplea el término colectivo *recursos*. El *workbench* de Eclipse puede verse como una macroherramienta que permite al usuario manipular un *workspace*, en base a un paradigma común para la creación, navegación y gestión en general de sus recursos.

Tipos de recursos y estructuración dentro de un workspace

Desde el punto de vista del usuario, existen tres tipos básicos de recursos:

- **Fichero**. Recurso de tipo no-contenedor. Los ficheros son análogos a los correspondientes del sistema de ficheros, conteniendo secuencias de bytes.
- **Carpeta**. Recurso de tipo contenedor. Las carpetas son análogas a los directorios del sistema de ficheros y pueden contener ficheros y otras carpetas, mientras que pueden estar contenidas en otras carpetas o en proyectos.
- **Proyecto**. Recurso de tipo contenedor. Los proyectos también se corresponden con directorios en el sistema de ficheros y contienen carpetas y ficheros, pero no otros proyectos

(aunque pueden tener visibilidad entre ellos), de forma que son los recursos de más alto nivel de contención en un *workspace*. Un proyecto se usa para organización de recursos relacionados con un área específica, para gestión de versiones, compartir, etc.

Las diversas herramientas que se conectan al *workbench* de Eclipse pueden definir sus propias especializaciones (naturalezas) de proyectos, carpetas y ficheros.

El almacenamiento y visualización de los recursos existentes en un *workspace* tiene lugar en forma de estructura jerárquica (árbol de contención), con proyectos en lo más alto y carpetas y ficheros debajo. Los conceptos básicos en tal jerarquía de recursos son:

- **Recurso padre**. Se aplica a cualquier recurso que contiene a otros recursos, por lo que sólo los proyectos y carpetas pueden serlo.
- **Recurso hijo**. Se aplica a cualquier recurso contenido dentro de otro, por lo que sólo los ficheros y las carpetas pueden serlo.
- **Raíz**. Recurso especial, que sirve como origen (raíz) del árbol de recursos, es decir, es el recurso de mayor nivel de contención en un *workspace*, y contiene como hijos inmediatos a los diferentes proyectos existentes. La raíz del *workspace* es creada internamente cuando éste se crea y existe mientras el *workspace* exista.

En el árbol de contención, cada nodo distinto de la raíz es de uno de los tres tipos básicos de recursos (incluyendo sus tipos especiales tratados en el siguiente apartado) y cada uno tiene un nombre distinto al de sus hermanos.

Recursos especiales

Todos los recursos de un proyecto residen en el mismo lugar del sistema de ficheros: el directorio correspondiente al proyecto. Entre ellos pueden existir carpetas y ficheros cuyo rol consiste en *representar vínculos* a ubicaciones en el sistema de ficheros (directorios y ficheros, respectivamente) fuera del proyecto. Estas carpetas y ficheros especiales se llaman *recursos vinculantes*[52] y suponen la posibilidad de "añadir" ficheros y carpetas a un proyecto que por alguna razón han de estar almacenados fuera de él. La ubicación fuera del proyecto no tiene que ser necesariamente externa también al *workspace*, de forma que los recursos vinculantes pueden utilizarse para que recursos de un proyecto puedan ser accesibles desde otro proyecto del mismo *workspace* (*overlapping* de recursos).

Los recursos vinculantes se comportan en la mayoría de aspectos como cualquier otro recurso en un *workspace*, pero sufren algunas restricciones en operaciones de tipo mover, copiar, etc.

Otro tipo especial de recurso son las *carpetas virtuales*. Se trata de carpetas que sólo existen en el árbol de recursos de un *workspace*, no correspondiéndose con ninguna localización física en el sistema de ficheros. Bajo una carpeta virtual pueden crearse otras carpetas virtuales o recursos vinculantes pero no recursos regulares, pues estos últimos necesitan como padre una localización real para poder existir en el sistema de ficheros.

[52] Las rutas hacia el artefacto *target* pueden definirse de forma absoluta o relativa a una variable de ruta. Las variables de ruta especifican localizaciones del sistema de ficheros.

Filtrado de recursos

A menudo, para reducir desorden y confusión, es necesario filtrar los contenidos mostrados por las vistas del *workbench* destinadas a la navegación por el *workspace*. Para ello es posible emplear diferentes técnicas:

- **Filtros**. Los filtros de recursos se emplean para filtrar ficheros por nombre (realmente extensión). Añadir filtros a un proyecto o carpeta permite, por un lado, seleccionar sistemáticamente entradas del sistema de ficheros para ser o no ser visualizadas en el árbol de recursos y por otro lado, configurar qué ficheros y carpetas en una jerarquía de recursos de un proyecto están automáticamente incluidos cuando se lleva a cabo un refrescado.
- **Conjuntos operativos (*working sets*)**. El *workbench* de Eclipse define el concepto de conjunto operativo como agrupador de elementos a efectos de su visualización en vistas o de la aplicación de operaciones en lote. Por ejemplo, las vistas navegacionales usan estos conjuntos operativos para restringir qué recursos son visualizados, incluyendo sólo recursos específicamente seleccionados. Existen *working sets* de diversos tipos: genérico, java, etc.

2.6.5 Propuesta de organización

Como ha quedado expuesto en la subsección anterior, el paradigma de gestión de recursos de Eclipse permite su organización mediante contenedores, pero no mediante etiquetado. Por tanto, la propuesta de organización de la información que se presenta en esta Tesis se basa en la estrategia de contenedores y se resume en los siguientes puntos:

- **Se adopta la noción de *espacio de trabajo* o *workspace* como solución de localización de la información**. Este punto es por congruencia con el empleo de Eclipse/EMF como plataforma de soporte a los entornos.
- **Dentro de un *workspace*, un determinado proyecto de desarrollo de SDTREs se representa mediante un proyecto Eclipse**.

  Un mismo *workspace* puede albergar diversos proyectos, de alguna manera relacionados, mientras que es posible la existencia simultánea de diversos *workspaces* para dar cobertura a proyectos de desarrollo independientes. Además, para mayor comodidad organizativa, pueden definirse *working sets* a conveniencia para conjuntos de proyectos muy estrechamente relacionados dentro de un *workspace*, como puede ser el caso de un proyecto de desarrollo de SDTREs y proyectos *plugin* que implementen infraestructura de soporte para él.
- **Dentro de un proyecto, se toma como primer criterio para clasificación y organización de la información el correspondiente a los metaniveles**, lo cual se plasma en dos carpetas al efecto: M1 y M2. Esta elección como criterio principal se debe a ser éste un criterio inherente a MDSE y por tanto exclusivamente aplicable a recursos MDSE.
- **A partir del segundo nivel de contención, el orden de aplicación de criterios clasificatorios es el siguiente**:

  En M1:

  1. El SDTRE bajo desarrollo.
  2. Fases del ciclo de desarrollo.
  3. Aspectos dentro de una fase.

4. Metodología/Herramientas. Cambios mayores en la versión de una herramienta se consideran herramientas distintas, mientras que cambios menores dan lugar a un nivel extra de anidamiento.
5. Modelos gestionados (nivel hoja).

En M2:

1. Fases del ciclo de desarrollo.
2. Aspectos dentro de una fase.
3. Metodología/Herramientas. Mismas consideraciones en cuanto a versionado.

   Los modelos correspondientes a cada metodología o entorno de herramientas son, a nivel M2, aquellos mediante los que se formaliza conceptualmente el dominio en cuestión, bien en forma de metamodelo, gramática, W3C-Schema, etc. Puesto que los paquetes de restricciones que se especifican para los metamodelos pueden encontrarse en recursos independientes, éstos también se ubican junto a sus respectivos metamodelos.

   Por otro lado, también se consideran recursos propios de una metodología determinada aquellas transformaciones de modelos, tanto internas a la metodología como las que la conectan con otra. Es por eso que en este nivel se consideran los subniveles:

   - Formalizaciones de dominio.
   - Transformaciones.

- **Por último, los niveles de anidación correspondientes a los criterios de TS y versionado se omiten**, delegando su especificación en el propio nombre del modelo, en concreto en la extensión y en un sufijo, respectivamente.

2.7 MAST-2: Ejemplo de representación de la información en un IDE

En el enlace http://www.istr.unican.es/members/cesarcuevas/phd/artifactsMAST2.html puede encontrarse todo el material desarrollado en el marco MAST-2 y relacionado con los contenidos de este capítulo. La Figura 2.19 y la Figura 2.20 muestran una visión esquemática de tales artefactos, los cuales ya han sido mencionados puntualmente a lo largo del capítulo.

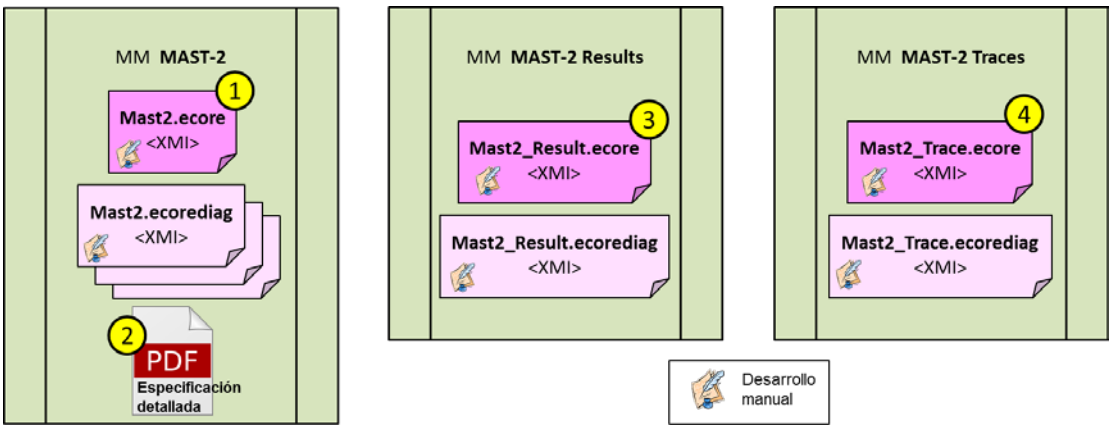


Figura 2.19 – Metamodelos del entorno MAST-2

1. **`Mast2.ecore`**. Formulación del metamodelo MAST-2 mediante el lenguaje Ecore.
2. **`Mast2_spec.pdf`**. Documentación detallada del metamodelo MAST-2.
3. **`Mast2_Result.ecore`**. Formulación del metamodelo MAST-2 Results mediante el lenguaje Ecore.
4. **`Mast2_Traces.ecore`**. Formulación del metamodelo MAST-2 Traces mediante el lenguaje Ecore.

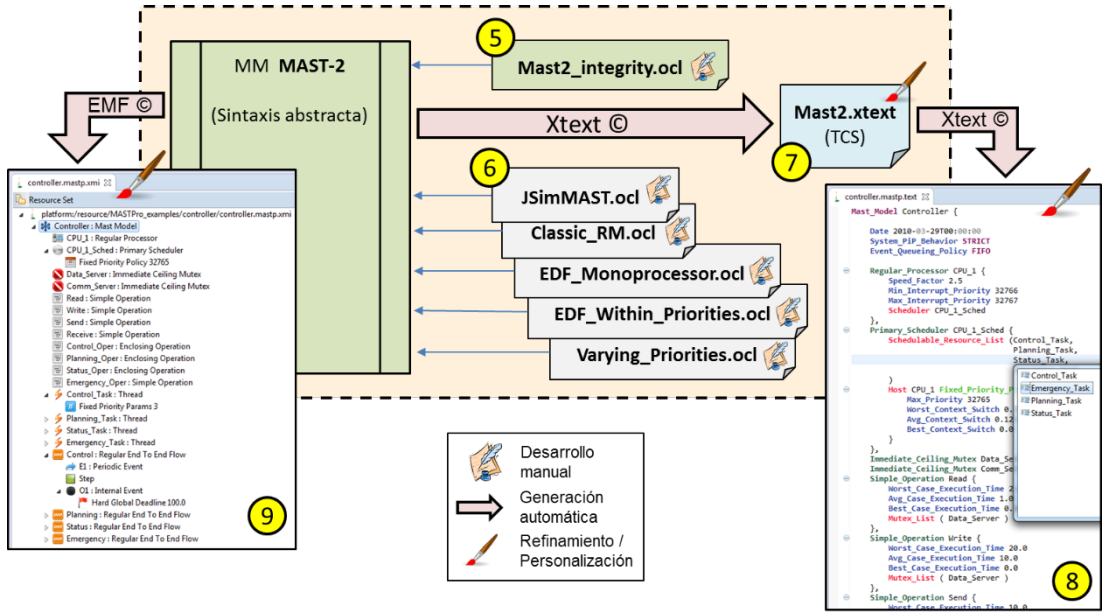


Figura 2.20 – Restricciones, gramática y editores para MAST-2

5. **`Mast2_integrity.ocl`**. Formulación de las restricciones de integridad especificadas sobre el metamodelo MAST-2 (*Restricciones MAST-2*) mediante OCL.
6. Formulación OCL de los diversos conjuntos de restricciones específicas de herramientas del entorno MAST.
7. **`Mast2.xtext`**. Formulación de la gramática MAST-2 mediante el *Xtext Grammar Language*.
8. **`TextualEditorMAST2.zip`**. *Plugins* Eclipse-Xtext que implementan el editor textual para modelos MAST-2.

Adicionalmente, empleando la potencia de EMF, se ha desarrollado un editor de tipo árbol especializado para MAST-2.

9. **`TreeEditorMAST2.zip`**. *Plugins* Eclipse-EMF que implementan el editor textual para modelos MAST-2.

3 Herramientas MDSE genéricas basadas en metaherramientas

La infraestructura de un entorno de desarrollo basado en MDSE, debe proporcionar tanto los recursos para contener y representar la información del sistema que se desarrolla como los recursos para crear, visualizar, actualizar, transformar y eliminar los modelos con los que se representa la información. En este capítulo se aborda la infraestructura y los recursos para realizar estas operaciones.

Un sistema software se describe mediante un gran número de modelos interreferenciados entre sí. Garantizar su consistencia global, mantener la sincronización horizontal y vertical, generar vistas apropiadas, aplicar patrones y reorganizar los modelos son tareas muy costosas que en la práctica sólo pueden realizarse si se dispone de herramientas para automatizar su gestión.

En el capítulo 1 se hizo una revisión de los recursos que proporciona la disciplina MDSE a este fin y que tienen como elemento central los lenguajes de transformación de modelos, ya sean declarativos, imperativos o híbridos. Sin embargo, a los ingenieros software que diseñan los entornos y sus herramientas no les resulta amigable su uso porque las encuentran ajenas al dominio en que son expertos. Esto es un hándicap que limita la consolidación de la metodología MDSE. En este capítulo se utiliza la propia estrategia MDSE para especificar, implementar y mantener las herramientas de transformación, en base a modelos que son amigables al responsable de su implementación porque corresponden a su dominio de conocimiento.

La contribución central del capítulo es la propuesta de herramientas genéricas que se formulan en base a metamodelos relacionados con su funcionalidad, y no en base a los metamodelos concretos de los modelos de entrada y salida sobre los que opera la herramienta, que es lo que habitualmente propone la metodología MDE.

Se analizan las diferentes formas de implementar herramientas genéricas, y en particular se propone una estrategia que consiste en desarrollar metaherramientas que incorporan en su código la funcionalidad que se quiere conseguir y que, en base a modelos de instrucción adaptan dicha funcionalidad a los modelos concretos sobre los que se opera en cada caso. En base a este modelo, la metaherramienta genera de forma automática y haciendo uso de la técnica HOT, la herramienta concreta que finalmente se aplica. De acuerdo con el papel de los agentes descrito en la sección 1.1, esta estrategia desplaza la responsabilidad de programación al agente *desarrollador de infraestructuras de entorno*, que sí es experto en lenguajes de transformación de modelos, y deja para el *diseñador de entornos de desarrollo* la tarea de formular los modelos de instrucción que sólo requieren el conocimiento en que éste último es experto. Asimismo, la estrategia propuesta facilita radicalmente el mantenimiento de las herramientas frentes a los cambios evolutivos de los metamodelos del entorno, lo que constituye uno de los principales problemas del desarrollo de herramientas en la metodología MDSE.

Como prueba de concepto de la estrategia, se proponen varios ejemplos de aplicación de la misma.

Visión general del capítulo

La **sección 3.1 (*Adaptación de los Entornos frente a la Evolución de su Ámbito Conceptual*)** considera el problema de la coevolución del ecosistema de artefactos relativos a un metamodelo y plantea enfoques para el desarrollo de herramientas genéricas, aplicables a modelos

conformes a diversos metamodelos y cuya implementación no se vea afectada ante la evolución de estos últimos.

La **sección 3.2 (*Herramienta para Verificación de Cumplimiento de Restricciones*)** plantea el desarrollo de una herramienta genérica implementada como una transformación M2M para verificar el cumplimiento de las restricciones que complementan un metamodelo laxo.

La **sección 3.3 (*Herramienta para Construcción de Modelos Acordes a Vistas de Dominio*)** plantea la especificación de vistas restrictivas sobre un metamodelo y el problema de construir modelos que además de ser conformes a sus metamodelos sean acordes a las vistas.

La **sección 3.4 (*Herramienta para Interoperabilidad XML ↔ Modelware*)** plantea una solución de interoperabilidad entre los espacios tecnológicos XML y *Modelware* para cuando las formalizaciones de un cierto dominio en ambos espacios hayan sido llevadas a cabo de manera independiente.

Por último, la **sección 3.5 (Material desarrollado)** presenta de una manera concisa el material desarrollado en relación a las tres metaherramientas presentadas en las secciones anteriores.

3.1 Adaptación de los Entornos frente a la Evolución de su Ámbito Conceptual

El dinamismo de la Ingeniería Software en general y la naturaleza polifacética de los entornos SDTRE en particular demandan una continua y costosa tarea de actualización de éstos. Ello hace que uno de los principales retos en su diseño sea dotarlos de mecanismos que posibiliten la flexibilidad necesaria para conseguir su adaptación ante la evolución en las áreas a las que han de dar cobertura. Tal evolución puede entenderse en dos dimensiones: relativa a los campos ya cubiertos o bien en términos de extensión hacia nuevos dominios. La MDSE tendrá opción de ser aceptada como base de los entornos sólo si facilita al ingeniero software (no experto en tecnologías MDSE) las tareas de actualización y mantenimiento de los mismos.

En entornos basados en MDSE, los dominios conceptuales cubiertos vienen representados por lenguajes de modelado descritos mediante metamodelos y las funcionalidades ofertadas se refieren a los procesos (principalmente transformaciones) que las herramientas de tales entornos pueden llevar a cabo sobre los modelos expresados mediante tales lenguajes, esto es, modelos conformes a dichos metamodelos.

En torno a cada metamodelo existe un conjunto de artefactos definidos en base a él y que se conoce como su ecosistema [116]. Éste comprende principalmente a sus modelos instancia y a las transformaciones de modelos en cuya definición participa (con rol de metamodelo fuente, destino o ambos), aunque también engloba otros tipos de artefactos, como por ejemplo restricciones de integridad, gramáticas o herramientas específicas para procesado de los modelos (visores, editores, generadores, etc.). La Figura 3.1 muestra una visión muy esquemática de un entorno basado en MDSE. Se limita a los metamodelos constituyentes de su ámbito conceptual y al conjunto de herramientas asociadas a cada uno. Por simplicidad, no se muestran otros artefactos como gramáticas o restricciones. La disposición de los metamodelos como base sobre la que se apoyan las herramientas ilustra la relación de dependencia existente.

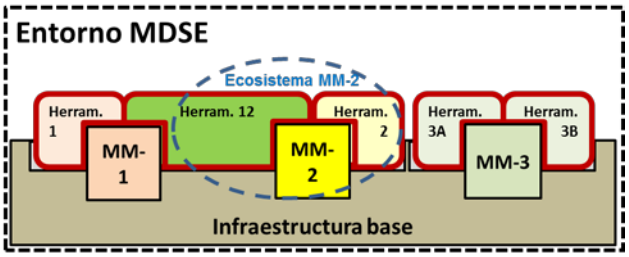


Figura 3.1 – Principales constituyentes de un entorno MDSE

Si la evolución de un entorno se reduce a la integración de nuevas herramientas o a la mejora de las ya presentes, no da lugar a efectos colaterales. En cambio, la evolución de un metamodelo puede afectar a su ecosistema de artefactos relacionados, planteando el problema de su adaptación. Desde el punto de vista de los entornos, el aspecto que resulta crítico es la adaptación de las herramientas [117]. La migración de otros artefactos, como gramáticas [118] o restricciones [119, 120], también puede serlo en tanto en cuanto sean parte de herramientas. En cambio, en principio la coevolución de los modelos terminales [121-124] no queda englobada en la del entorno que los gestiona, al no ser éstos elementos constitutivos de él.

La otra dimensión en que un entorno puede evolucionar es expandiéndose hacia nuevos dominios, lo cual en el caso de entornos MDSE significa incorporación de nuevos metamodelos. La dimensión anteriormente expuesta, evolución del entorno por evolución de los metamodelos existentes, podría considerarse un caso particular de ésta segunda, pues en el fondo también se da lugar a nuevos metamodelos. Sin embargo, la diferencia radica en que mientras que el primer caso supone la revisión y adaptación de las herramientas ya existentes, en general, la incorporación de nuevos metamodelos no implica ningún tipo de coevolución, sino que conlleva la creación de nuevas herramientas. Éstas pueden ser análogas a otras ya presentes relativas a otros dominios o completamente nuevas.

En cualquier caso, tanto el efecto que supone sobre las herramientas de un entorno la evolución de metamodelos como el esfuerzo requerido para equiparlo con nuevas herramientas de soporte debido a la incorporación de nuevos metamodelos, se deben fundamentalmente a que las herramientas típicamente obedecen a un escenario básico: su funcionalidad se corresponde únicamente con un determinado dominio, de forma que en su especificación e implementación se hace referencia explícita al metamodelo al que son conformes los modelos que procesan. A este tipo de herramienta la denominamos ***herramienta MDSE estándar*** y, dado que su implementación está ligada a la estructura del correspondiente metamodelo, resulta inmediato advertir las siguientes dificultades en el mantenimiento de un entorno:

- En general, estas herramientas pueden verse afectadas por la evolución de los metamodelos respecto a los que se han diseñado, requiriéndose su revisión.
- La incorporación de nuevos metamodelos al entorno requiere el desarrollo *ad hoc* de las correspondientes herramientas.

Para evitar estos hándicaps, en esta Tesis se propone que los entornos hagan un uso extensivo de *herramientas genéricas* capaces de operar sobre modelos conformes a diferentes metamodelos[53]. Las subsecciones a continuación profundizan en ambos tipos de herramientas.

[53] Esto no significa modelos que cada uno de ellos es conforme a más de un metamodelo, lo cual carece de sentido salvo si se considera un metamodelo y sus variantes semánticamente equivalentes o el caso de herencia de

3.1.1 Herramientas MDSE estándar

En esta Tesis se utiliza el término *herramienta MDSE estándar* para referirse a aquellas herramientas que especifican e implementan su funcionalidad en base a un determinado metamodelo, de forma que únicamente son capaces de actuar sobre modelos conformes a él. La Figura 3.2 esquematiza este escenario mostrando la limitación de una herramienta MDSE estándar concebida para procesar modelos conformes a un cierto metamodelo (MM-2). Puesto que es específica para él, no acepta como entrada modelos conformes a ningún otro metamodelo (MM-1 y MM-3 en la Figura 3.2).

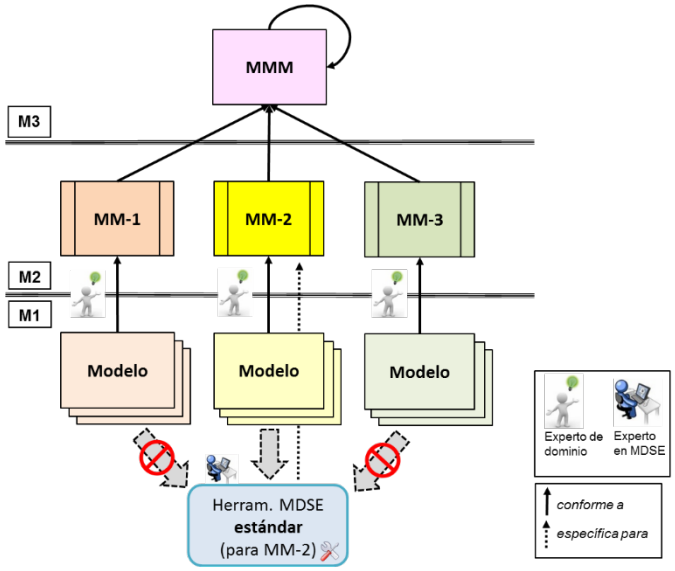


Figura 3.2 – Herramienta MDSE estándar

La construcción de la herramienta es responsabilidad de un desarrollador experto en infraestructura MDSE a partir de la especificación propuesta por un experto en el dominio en cuestión. Dado que su implementación es dependiente del metamodelo, cualquier modificación en la estructura de éste implica la necesidad de adaptación del código, requiriendo de nuevo la participación del experto MDSE y dificultando con ello el mantenimiento del entorno. Además, la extensión del entorno a nuevos dominios incorporando nuevos metamodelos exige el desarrollo de nuevas herramientas estándar asociadas a ellos, planteando similares condicionantes.

3.1.2 Herramientas MDSE genéricas

En esta Tesis se utiliza el término *herramienta MDSE genérica* para referirse a aquellas herramientas capaces de llevar a cabo una cierta funcionalidad (procesamiento) sobre modelos conformes a diferentes metamodelos. En algunos casos puede tratarse de herramientas capaces de operar sobre modelos con independencia de cuál sea su metamodelo (*herramienta MDSE universal*[54]), aunque en general se trata de herramientas que pueden aplicarse a una familia de metamodelos relacionados por ciertas características comunes.

metamodelos (ver subsección 3.2.3). Significa diversos metamodelos y los correspondientes conjuntos de modelos conformes, cada uno de ellos sólo conforme a un metamodelo.

[54] Universalidad dentro de los límites del metametamodelo único que da soporte al *framework* de modelado empleado.

La Figura 3.3 muestra una herramienta genérica aplicable tanto a los modelos conformes al MM-1, como a los conformes al MM-2 y también a los conformes al MM-3. En el escenario mostrado, la herramienta no llega a poseer carácter universal, pues existen dominios, como el descrito por el MM-n, para los que no es capaz de procesar sus modelos.

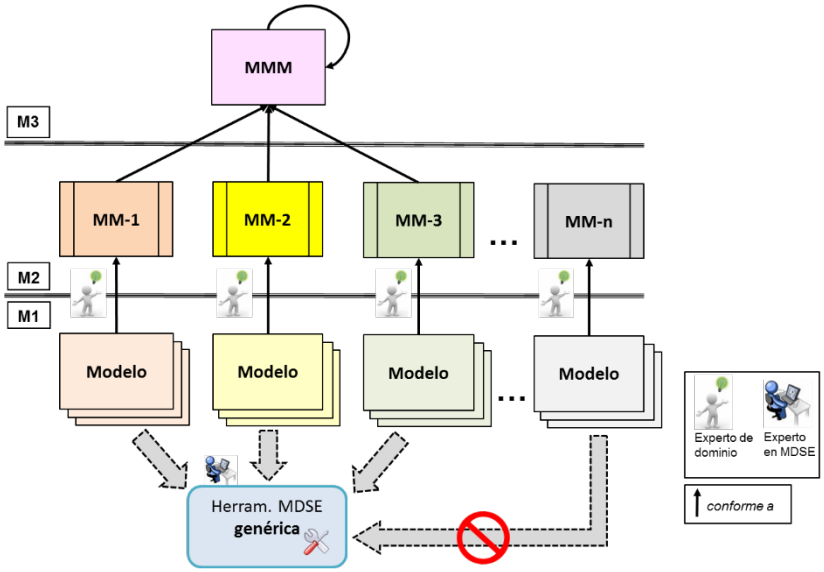


Figura 3.3 – Herramienta MDSE genérica

Para que una herramienta tenga carácter genérico es necesario que su funcionalidad sea relativa a aspectos pertenecientes a un *espacio conceptual común* (ECC), compartido por los diversos metamodelos a los que se pretende dar cobertura.

3.1.3 Algunos escenarios de espacio conceptual común.

En esta subsección se presentan algunos escenarios de ECC sobre los que basar el desarrollo de herramientas genéricas. Algunos posibilitan el desarrollo de herramientas universales, mientras que otros únicamente el desarrollo de herramientas genéricas acotadas.

El metametamodelo

Dentro del contexto de un determinado *framework* de modelado (como EMF), pueden desarrollarse herramientas cuya funcionalidad únicamente requiera procesar información de los modelos definida a nivel del metametamodelo (único) núcleo del *framework* (como Ecore para EMF). En este caso se trataría de herramientas genéricas universales, capaces de procesar uniformemente los modelos conformes a cualquier metamodelo, siendo el propio metametamodelo quien desempeña el rol de ECC. La Figura 3.4 ilustra este enfoque, mostrando una herramienta genérica universal cuya funcionalidad está basada en conceptos pertenecientes al metametamodelo.

Un ejemplo trivial de este tipo de herramienta sería una que analice un modelo cualquiera y cuente los objetos contenidos, no considerados como instancias de las clases definidas en el correspondiente metamodelo, sino como instancias de EObject (considerando el contexto EMF/Ecore).

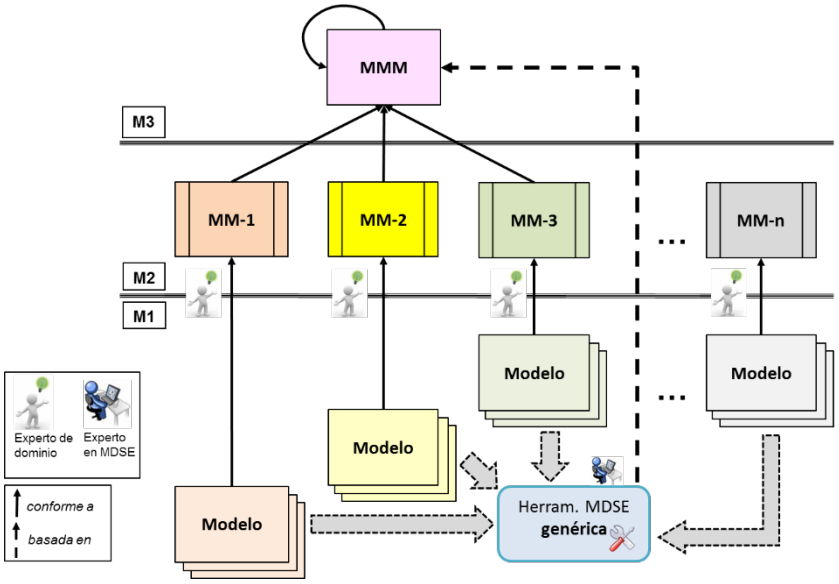


Figura 3.4 – Herramienta MDSE genérica (universal) basada en el metametamodelo

Otro ejemplo algo más sofisticado es nuestra herramienta *ModelWatcher* mostrada en la Figura 3.5, que permite la visualización en formato árbol de cualquier modelo, independientemente de su metamodelo. Lógicamente, emplea iconos genéricos para mostrar los objetos, los valores asignados a atributos y el establecimiento de referencias.

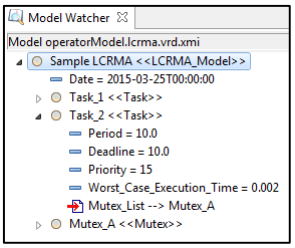


Figura 3.5 – Visor *ModelWatcher*

En el desarrollo de esta Tesis se han empleado herramientas basadas en el metametamodelo pero no son consideradas como contribución. De hecho, en EMF se encuentran disponibles multitud de herramientas universales. Por ejemplo:

- El *Sample Reflective Ecore Model Editor*.
- La herramienta de chequeo de conformidad de un modelo respecto a su metamodelo.

Metamodelo extendido

A partir del concepto de herencia entre clases propio de la OO puede definirse la noción de que un metamodelo sea extendido por otro. En esta situación, el metamodelo extensor tiene visibilidad sobre el extendido y existen relaciones de herencia entre las clases de ambos. Por ejemplo, que la clase contenedor principal del metamodelo extensor herede de la clase contenedor principal del metamodelo extendido. Si esta situación engloba un conjunto de metamodelos extendiendo al mismo metamodelo inicial (una familia de metamodelos extensores), los modelos conformes a cualquier miembro de la familia contienen información cuya descripción se encuentra en un ECC: el metamodelo extendido. Así, una herramienta cuya funcionalidad consista en procesar esa parte de la información sería de hecho una herramienta genérica, capaz de ser aplicada sobre modelos conformes a diferentes metamodelos (los que componen la familia extensora). La Figura 3.6 ilustra este escenario.

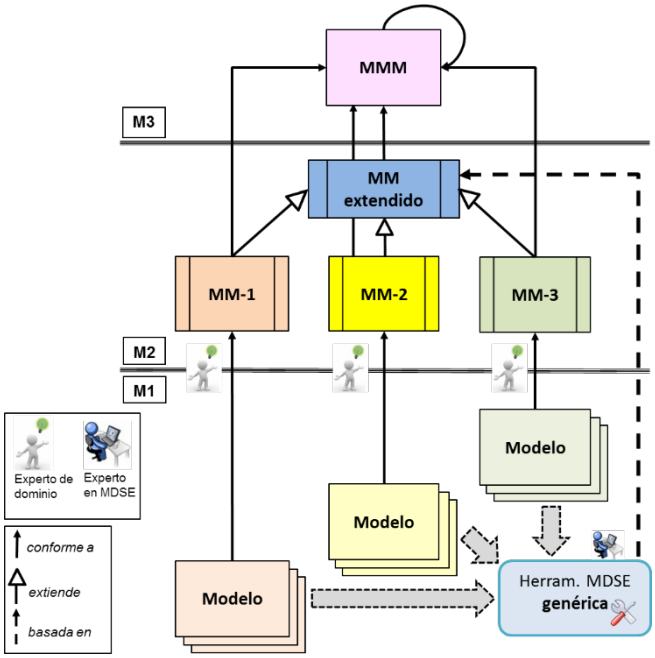


Figura 3.6 – Herramienta MDSE genérica basada en extensión de metamodelo

Metamodelo de instrucción

Los escenarios expuestos permiten la construcción de herramientas genéricas con código estático y que no requieren de ningún tipo de información suplementaria que posibilite su adaptación a cada uno de los metamodelos a los que dar cobertura. Sin embargo, también es posible un escenario en que sea necesario suministrar a la herramienta genérica una información adicional por cada metamodelo al que dar soporte. Tal información *instruye* a la herramienta en cómo adaptarse para ampliar su cobertura al nuevo metamodelo y poder así procesar modelos conformes a él. En este caso, el ECC se corresponde con el dominio conceptual al que pertenece la información instructora. Si ésta se formula como modelo (*instructor*), el ECC es representado por el metamodelo de los modelos instructores. Denominamos *metamodelo de instrucción* a este metamodelo asociado a la herramienta. La Figura 3.7 ilustra este escenario.

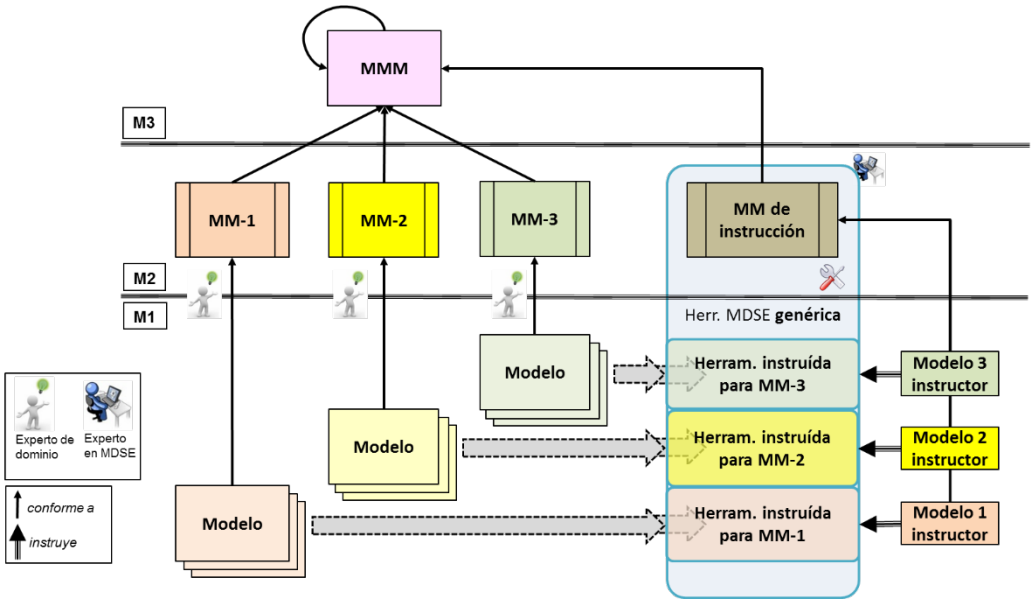


Figura 3.7 – Herramienta MDSE genérica basada en metamodelo de instrucción

Por tanto, este enfoque implica desarrollar un modelo instructor específico relativo a cada metamodelo al que dar soporte. En función de cómo la herramienta utilice los modelos instructores, se puede distinguir entre *herramientas basadas en configuración* y *herramientas basadas en metaherramienta*.

- **Herramienta basada en configuración**. Herramienta con código estático que se configura en su ejecución a partir de la información proporcionada por un modelo instructor, el cual hace referencia al correspondiente metamodelo de dominio. La Figura 3.8 ilustra este escenario.

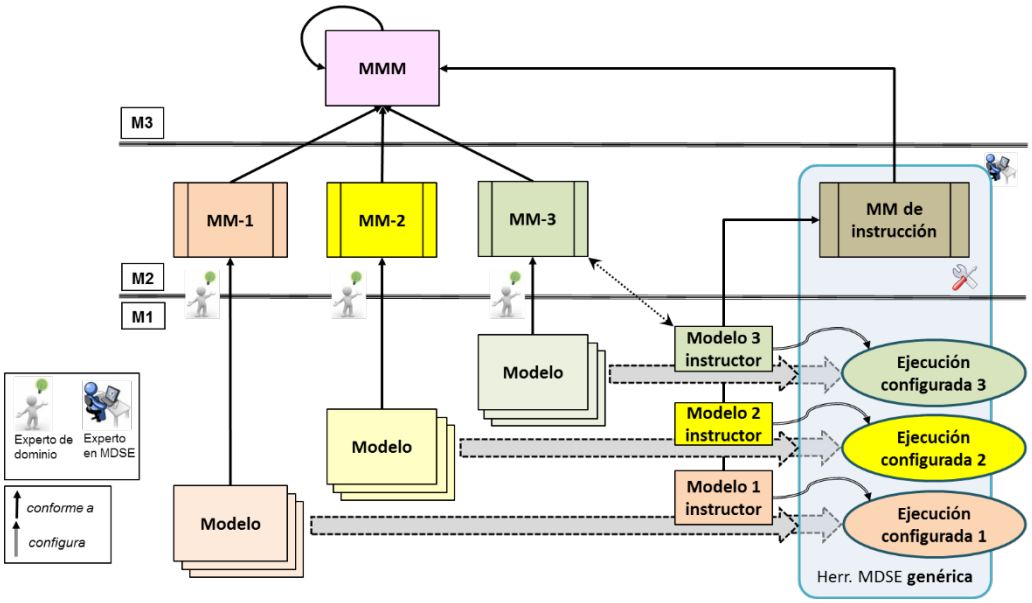


Figura 3.8 – Herramienta MDSE genérica basada en configuración

Esta estrategia es conceptualmente simple pero poco escalable. Cuando las posibilidades de configuración crecen, el código de la herramienta se hace complejo y difícil de mantener.

- **Herramienta basada en metaherramienta**. Herramienta con código estático capaz de generar herramientas con una determinada funcionalidad pero cada una orientada a un determinado dominio (metamodelo). Básicamente, una metaherramienta opera en dos pasos:

  1. Generación de la herramienta específica con la funcionalidad que le corresponde (en caso de que no haya sido anteriormente generada).
  2. Aplicación de la herramienta generada a modelos conformes al metamodelo previsto por el modelo de instrucción.

  De esta forma, la metaherramienta funciona como una herramienta genérica capaz de aceptar como entrada cualquier modelo sin importar su metamodelo. La Figura 3.9 ilustra este escenario.

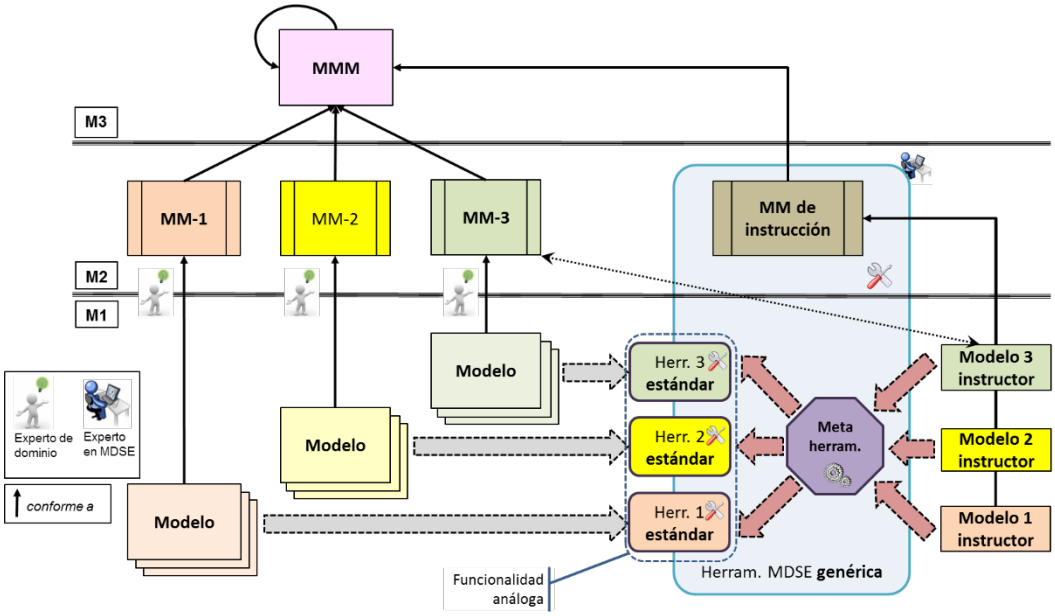


Figura 3.9 – Herramienta genérica basada en metaherramienta

Esta estrategia resuelve el problema de la coevolución, porque hay una gran diferencia entre tener que desarrollar el código de una herramienta específica o tener que formular un simple modelo de instrucción. Lo primero requiere la intervención de un experto en infraestructura MDSE mientras que lo segundo puede ser realizado por el propio experto de dominio. La participación del experto MDSE sólo es necesaria para desarrollar *una única vez* la herramienta genérica instruíble, incluyendo el diseño del metamodelo de instrucción.

En esta Tesis se ha apostado por adoptar este último enfoque en su vertiente de metaherramienta para equipar a un entorno con herramientas genéricas. Con el propósito de hacerlo viable en la práctica, se ha ideado una estrategia para construcción de metaherramientas. La siguiente subsección expone tal estrategia.

3.1.4 Estrategia para construcción de metaherramientas

Puesto que la característica básica de las metaherramientas es su capacidad para generar herramientas estándar específicas, al abordar el diseño de una estrategia para su desarrollo es necesario establecer algún tipo de catalogación de las herramientas que se desea producir. Dado que éstas van destinadas a entornos basados en MDSE, sus componentes nucleares serán transformaciones M2M, la operación más importante en MDSE. Por tanto, en esta Tesis la atención se centra en funcionalidades (procesamientos de modelos) que sea posible identificar o interpretar en clave de transformación M2M.

En la subsección 1.2.3 ya se dedicó un apartado a las transformaciones de modelos. Como breve recordatorio, una transformación M2M toma, en su concepción más general, 1..M modelos de entrada (cada uno conforme a un metamodelo dentro de un conjunto de metamodelos fuente) y genera 1..N modelos de salida (cada uno conforme a un metamodelo dentro de un conjunto de metamodelos destino, entre los cuales pueden encontrarse metamodelos fuente). Se recuerda también que aunque habitualmente los modelos tanto de entrada como de salida corresponden a artefactos en la capa M1 (modelos terminales), en general éstos pueden

contener referencias a elementos de metamodelos o incluso tratarse directamente de metamodelos.

Una vez establecido que las herramientas a producir consistan en transformaciones M2M, el diseño de la estrategia para desarrollo de metaherramientas ha de estar guiado por el objetivo de la generación automática de tales transformaciones. Así, dado el metamodelo de instrucción asociado a una metaherramienta, ésta ha de crear automáticamente la herramienta M2M correspondiente a cada modelo instructor. Por tanto, sólo resta elegir una técnica para implementar la generación automática de transformaciones. En esta Tesis se ha escogido la técnica de Transformaciones de Orden Superior (HOT) y a ella se dedica el siguiente apartado.

Técnica HOT

Uno de los aspectos más interesantes relacionados con la visión *todo es un modelo* que propugna MDSE es que las propias transformaciones de modelos pueden ser vistas y gestionadas como modelos [125], incluyendo su metamodelado. Así, definir un metamodelo para modelar transformaciones, es decir, recoger en forma de metamodelo los conceptos y reglas que rigen su definición, permite que cualquier transformación pueda expresarse como un modelo conforme a él, que se llamaría *modelo de transformación*. Tales modelos de transformación pueden ser manejados por medio de otras transformaciones, es decir, pueden ser entrada o salida de otras transformaciones de modelos.

Sobre una infraestructura en la que sea posible modelar las propias transformaciones de modelos, se define una HOT como:

> *Una transformación de modelos que consume o produce modelos de transformación, es decir, sus modelos de entrada y/o salida son ellos mismos (modelos de) transformaciones de modelos.*

Por tanto, una HOT toma uno o más modelos de transformación como entrada, produce uno o más como salida o ambas cosas, permitiendo así aplicar los principios del MDSD al desarrollo de transformaciones (construir transformaciones complejas componiendo varias transformaciones más simples, soportar la evolución de metamodelos y la coevolución de modelos, definir cadenas de transformaciones, reutilizar transformaciones existentes, etc.).

Para que la aplicación de esta técnica sea factible es necesario disponer de un lenguaje de transformación cuya sintaxis esté formalizada como metamodelo, de manera que las distintas transformaciones implementadas en dicho lenguaje puedan representarse como modelos conformes a él. Por otro lado, también es necesario que tal lenguaje cuente con mecanismos que posibiliten la conversión de una transformación en formato textual a modelo y viceversa.

En esta Tesis, el empleo de ATL como lenguaje de transformación de modelos satisface las necesidades mencionadas, pues la plataforma *ATLAS Model Management Architecture* (AMMA) [73], cuna de ATL, proporciona el metamodelo `ATL.ecore` así como sendos mecanismos *ATLAS Technical Projector* (ATP) mediante los que convertir una transformación en formato textual (documento *.atl) a modelo ATL (inyección ATL) y serializar un modelo ATL a la sintaxis textual concreta (extracción ATL). La Figura 3.10 muestra el escenario expuesto mientras que la Figura 3.11 muestra el núcleo del metamodelo de ATL.

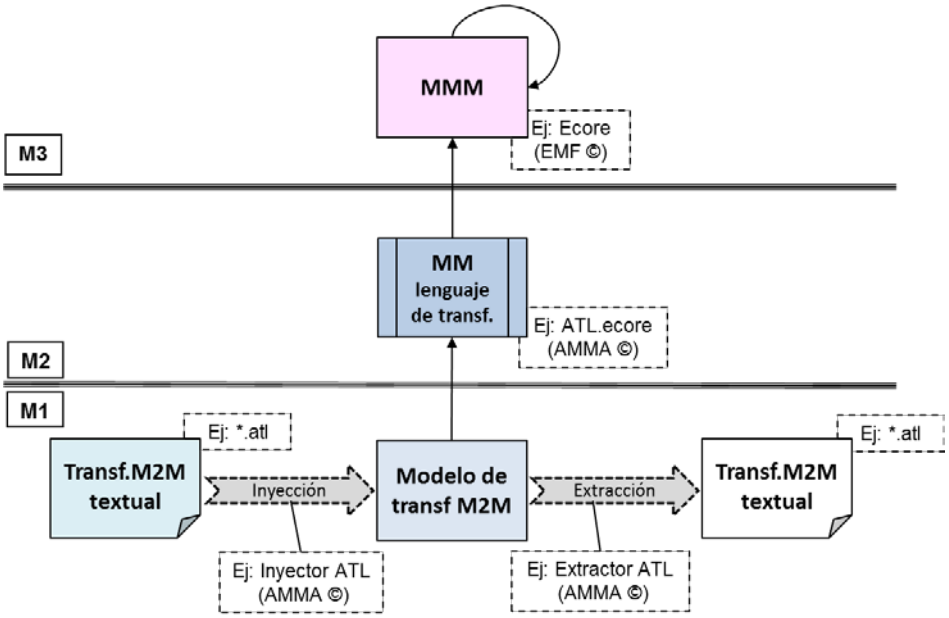


Figura 3.10 – Transformación M2M vista como artefacto textual y como modelo

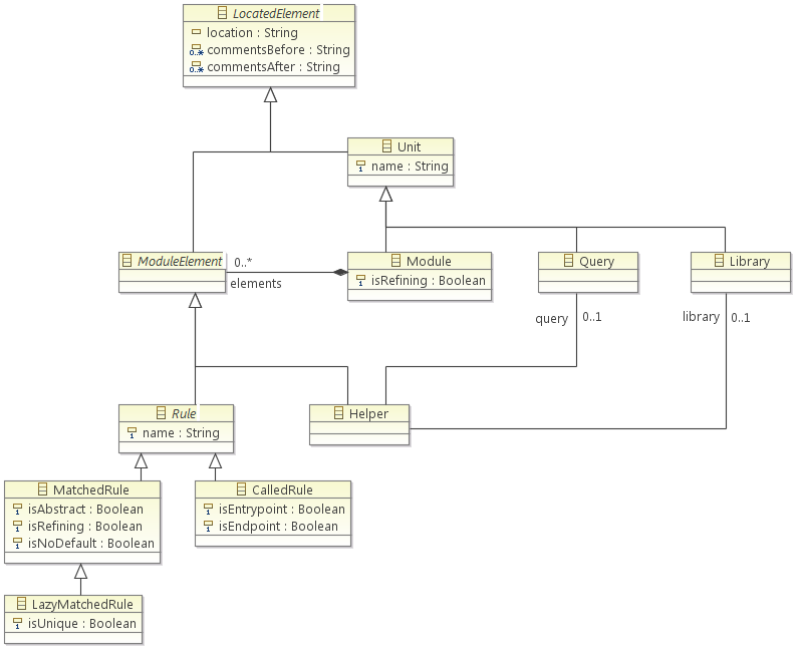


Figura 3.11 – Núcleo del metamodelo ATL

Las HOT en que se basan las metaherramientas presentadas en las siguientes secciones responden al patrón de *HOT de síntesis*, que se define como aquel correspondiente a las HOT que generan una (un modelo de) transformación a partir de modelos que no representan transformaciones, es decir, el modelo de salida corresponde a una transformación pero los modelos de entrada no.

Una visión en profundidad sobre la técnica HOT, los patrones de uso en que se pueden clasificar las HOT, los procedimientos bajo los cuales se realiza la conversión entre transformaciones textuales y modelos de transformación así como los entornos que proporcionan soporte para modelado de transformaciones puede encontrarse en [126, 127].

3.2 Herramienta para Verificación de Cumplimiento de Restricciones

En la sección 2.3 se ha justificado la formulación de metamodelos laxos complementados con la especificación de restricciones de integridad inherentes a la semántica del dominio conceptual cubierto. En estos casos, la formalización de un dominio ya no consiste solamente en el metamodelo sino también en el conjunto de restricciones, de manera que la conformidad de los modelos engloba tanto su conformidad básica respecto al metamodelo como el cumplimiento de cada restricción. Asimismo, cada herramienta encargada de procesar los modelos puede introducir un nuevo conjunto de restricciones para delimitar las características de los modelos que son exigidas por su naturaleza y alcance. De esta forma, se puede emplear un único metamodelo para formalizar la información del dominio en el entorno, aunque sólo algunos de sus modelos podrán ser procesados por una determinada herramienta. E incluso si el metamodelo y la herramienta estuvieran totalmente alineados, ésta podría imponer restricciones debido a su estado de desarrollo o limitaciones por su implementación.

Por tanto, el cumplimiento de un conjunto de restricciones por parte de un modelo determina su conformidad, en caso de tratarse de restricciones intrínsecas, o dictamina su validez para ser procesado por una cierta herramienta, en caso de ser restricciones específicas a ella. En cualquier caso, el problema de la verificación del cumplimiento de restricciones por parte de los modelos es el mismo y ha de ser abordado mediante herramientas al efecto, construidas en base a las propias restricciones en cuestión.

Motivación y conveniencia de la herramienta genérica contribuida

Esta sección contribuye al desarrollo de herramientas para validación de modelos. En concreto, se aborda el diseño de una herramienta genérica para verificación del cumplimiento de restricciones. Si el propio entorno ofrece una herramienta genérica para ello, las restantes herramientas quedan notablemente aligeradas al poder simplemente invocarla y no tener que implementarla en su código. El soporte a la verificación de restricciones que ofrece EMF se considera insuficiente porque es muy críptico. Se limita a una ventana emergente que confirma que el modelo satisface las restricciones o proporciona una lista con los incumplimientos, descritos por el identificador de la restricción violada y el elemento del modelo en el que se incumple.

La herramienta genérica que se propone implementa una estrategia *model-driven* pura. Se formula como una transformación M2M en la que las entradas son el modelo que se verifica junto al modelo que describe las restricciones y la salida es un modelo que describe los incumplimientos detectados. El metamodelo del modelo que se verifica puede ser cualquiera, mientras que los metamodelos de soporte a los modelos que describen las restricciones y las violaciones están definidos como parte de la herramienta.

Si la estrategia se implementase mediante herramientas MDSE estándar, sería necesario el desarrollo *ad hoc* de una herramienta específica para cada pareja de metamodelo y paquete de restricciones, debiendo además ser convenientemente adaptada ante cualquier modificación en éstas últimas y ante cualquier modificación del metamodelo que a su vez influya en su especificación. Como se ha esbozado en la sección 3.1, ésta sería una solución no deseable, pues:

- Dado un metamodelo y sus conjuntos de restricciones asociados, aunque éste se mantenga inalterado:

- Las herramientas encargadas de procesar los modelos pueden evolucionar. El paso a nuevas versiones típicamente implica una reducción en el conjunto de restricciones específicas, lo que implicaría el rediseño de la correspondiente herramienta de verificación.
- Nuevas herramientas pueden ser desarrolladas, cada una probablemente aportando un nuevo paquete de restricciones, lo que conllevaría tener que desarrollar una nueva herramienta específica a cada caso.
- Nuevas laxitudes pueden ser detectadas en el metamodelo. Esto implica la ampliación del conjunto de restricciones intrínsecas y la consiguiente adaptación de la herramienta de verificación.

- Si además el metamodelo evoluciona, esto puede tener diversas consecuencias sobre las restricciones especificadas, tanto intrínsecas como específicas:
  - En general, puede ser necesaria su reformulación.
  - Puede producirse la desaparición de algunas laxitudes y la aparición de otras nuevas. De nuevo, esto implica la modificación del conjunto de restricciones intrínsecas.

  En ambos casos se hace necesario el rediseño de las correspondientes herramientas de verificación.
- La incorporación de un nuevo metamodelo a un entorno podría requerir la construcción de nuevas herramientas de verificación.
  - Una para el conjunto de restricciones intrínsecas, en caso de ser un metamodelo laxo.
  - Una por cada paquete de restricciones adicional definido.

Por todo ello, se propone una herramienta genérica para verificación, basada en una metaherramienta que bajo demanda genere cada herramienta específica de verificación e invoque su ejecución. Éstas van a adoptar la forma de transformación M2M y, por tanto, la metaherramienta está basada en una HOT.

Herramienta basada en metaherramienta

La Figura 3.12 muestra los elementos básicos de la herramienta diseñada. La caracterización y diseño como herramienta genérica requiere la definición del ECC que describe su funcionalidad y su consiguiente formalización como metamodelo de instrucción. El ECC se tratará en una subsección posterior, aunque para facilitar la comprensión de la Figura 3.12 se indica que el *metamodelo CC* (*Constraints Characterization*) constituye el metamodelo de instrucción y cada modelo instructor instancia suya caracteriza un cierto conjunto de restricciones.

Dado un conjunto de restricciones especificado sobre un metamodelo, la formulación de un modelo de caracterización de tal conjunto instruye a la metaherramienta (basada en HOT), núcleo de la herramienta genérica de verificación, para que produzca la herramienta específica adecuada (basada en M2M).

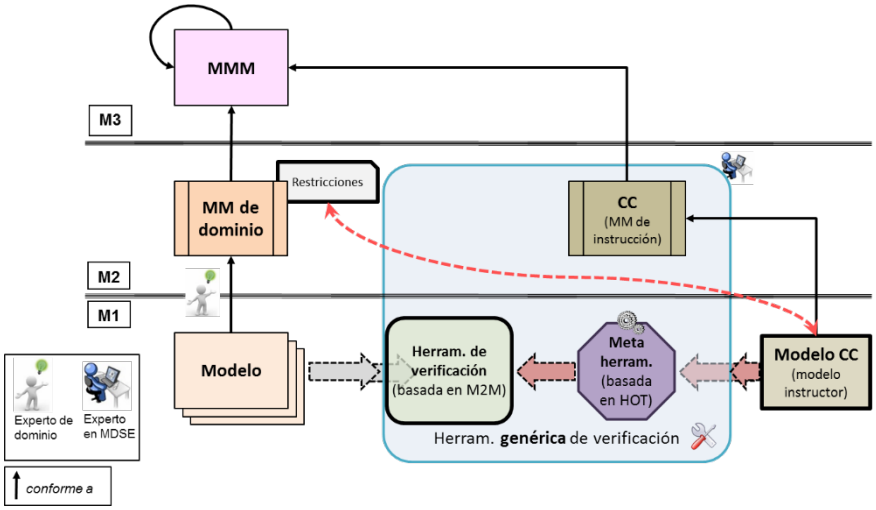


Figura 3.12 – Esquema de la herramienta genérica de verificación

La Figura 3.13 incorpora nuevos detalles a la Figura 3.12 a fin de resaltar el carácter universal de la herramienta diseñada. Dado *cualquier* conjunto de restricciones especificado sobre *cualquier* metamodelo, la formulación del correspondiente modelo instructor permite la obtención de la correspondiente herramienta específica.

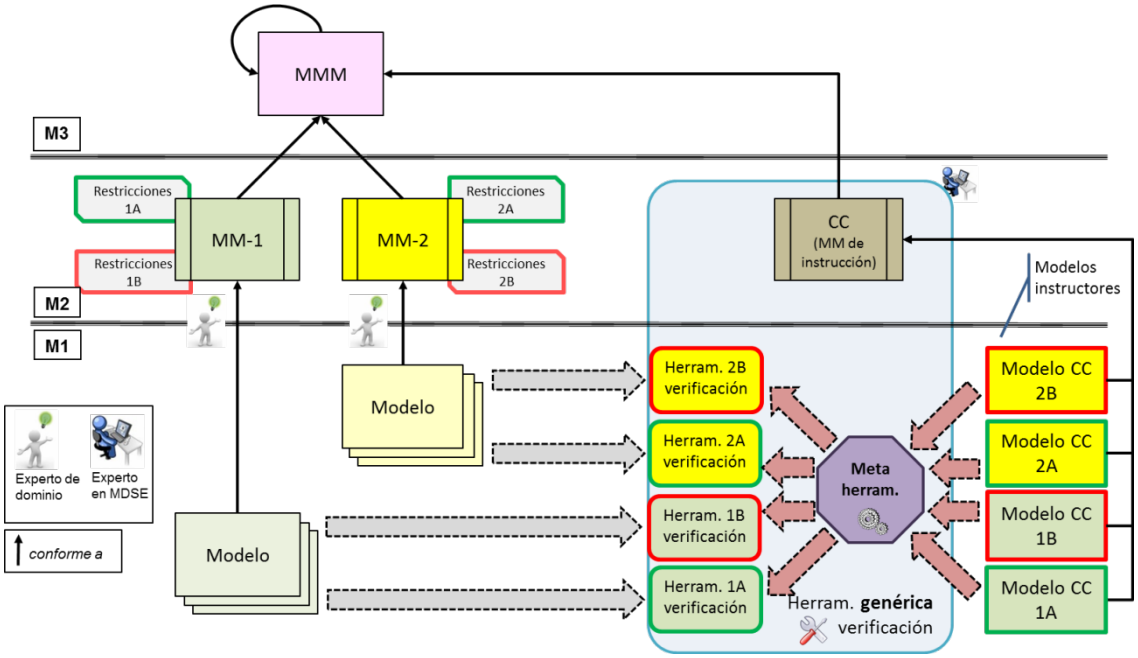


Figura 3.13 – Naturaleza universal de la herramienta diseñada

Así, la actuación ante cualquier supuesto dentro de la casuística evolutiva enumerada anteriormente se reduce a la formulación del modelo instructor que caracteriza al nuevo (renovado o inédito) conjunto de restricciones. A partir de este modelo se genera la nueva herramienta específica de verificación, sin necesidad de adaptación de la que ha quedado obsoleta, la cual puede desecharse.

3.2.1 Resultado de la verificación en forma de modelo

El resultado de verificar un modelo puede tomar diferentes formas. Por ejemplo, la forma más simple es un valor booleano que indica si el modelo satisface todas las restricciones (verdadero) o si incumple al menos una (falso). También puede adoptar la forma de un valor entero que indique cuántas restricciones han sido incumplidas o incluso un valor enumerado. Aquí se ha apostado por proporcionar el resultado de la verificación en forma de modelo. Esta idea, ilustrada por la Figura 3.14, y la consiguiente de concebir el proceso de verificación como transformación M2M, ya fueron esbozadas por Bézivin en [128]. Representar el resultado de la verificación en forma de modelo permite su participación en posteriores procesos *model-driven*, con propósitos que pueden ir desde la manifestación detallada de los incumplimientos cometidos hasta su corrección automática, etc.

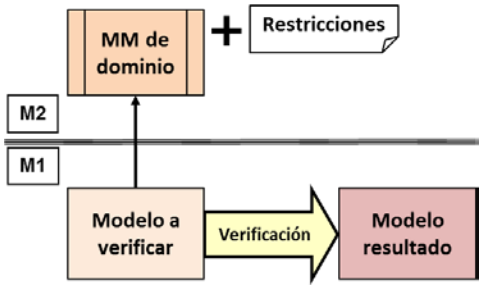


Figura 3.14 – Resultado de la verificación de un modelo en forma de otro modelo

Cada elemento del modelo de salida representa el incumplimiento de una restricción por parte de un elemento del modelo verificado y formula información descriptiva sobre tal incumplimiento, con la profundidad de detalle que se establezca (origen, naturaleza, severidad, etc.). El hecho de que en tales modelos-resultado se formulen los datos necesarios para describir los incumplimientos detectados les permite constituir la base para la posible manifestación de éstos (por ejemplo, en forma de mensajes textuales). Además, como modelos, pueden ser manejados por parte de las herramientas de un entorno dentro de un proceso MDSE.

La completitud de este enfoque requiere de un metamodelo que dé soporte a los modelos resultantes de la verificación. Se propone denominarlo metamodelo para *Descripción de Violaciones de Restricciones* (CVD). La Figura 3.15 completa a la Figura 3.14 con esta perspectiva.

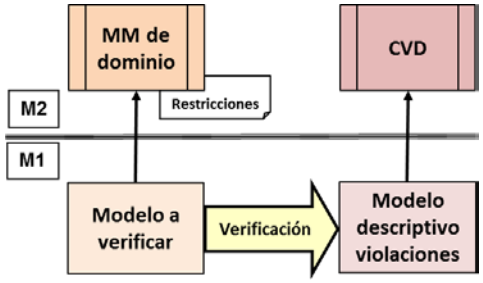


Figura 3.15 – Modelo resultado de la verificación y conforme al metamodelo CVD

3.2.2 Metamodelo CVD

Este metamodelo constituye una propuesta de metamodelo para formalización de los modelos mediante los cuales formular el resultado de verificar otros modelos y aspira a ser capaz de cubrir todo el espectro de violaciones de restricciones que puedan darse, permitiendo modelar

con mayor o menor detalle los datos necesarios para describir tales incumplimientos. Para ello, presenta una jerarquía de clases que posibilita dicho modelado, incremental en cuanto a detalle, de manera tanto más detallada cuanto más se profundiza en la jerarquía.

El metamodelo CVD presenta una estructura convencional, con una clase contenedor principal (`CVD_Model`) y una clase raíz (`CVD`) de la que heredan el resto de clases, conformando en conjunto la anteriormente mencionada jerarquía de clases destinadas al modelado de (los datos necesarios para describir) incumplimientos de restricciones.

La Figura 3.16 muestra las clases contenedor principal y raíz, así como las principales subclases de ésta última.

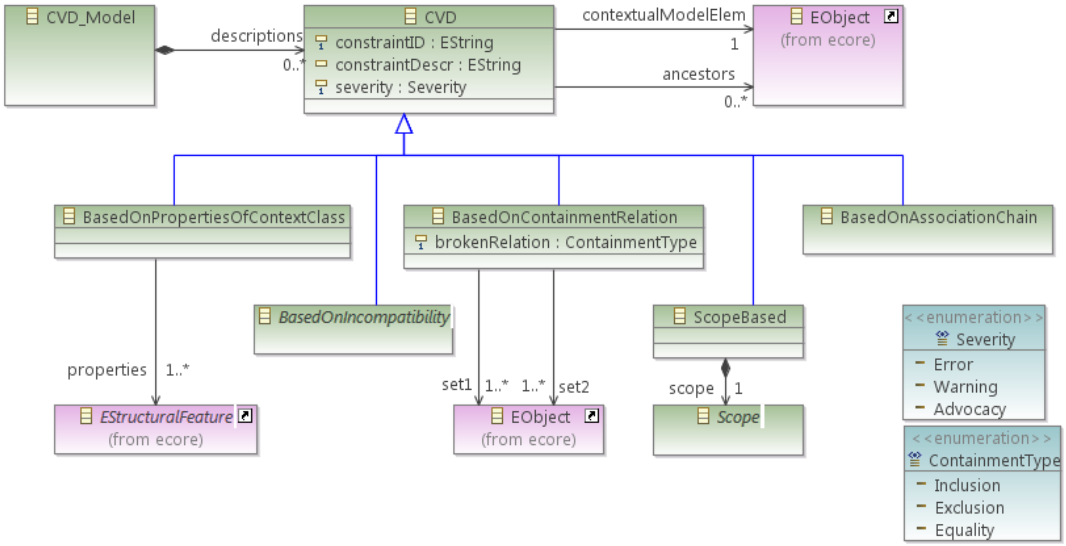


Figura 3.16 – Núcleo del metamodelo CVD

Un modelo conforme a CVD posee una única instancia de `CVD_Model`, la cual contiene a través de la asociación-composición `descriptions` al resto de elementos de modelo, instancias de `CVD` o de cualquiera de sus subclases.

A continuación se describen brevemente la clase raíz `CVD` y sus subclases de primer nivel:

- **`CVD`**. Esta clase permite modelar de forma genérica la descripción de incumplimientos de restricciones de cualquier tipo.

  Únicamente con esta clase raíz sería suficiente para formular como modelo el conjunto de incumplimientos de restricciones detectados en un modelo, pues la información que describe es apta para cualquier restricción, independientemente de su naturaleza, semántica o formulación OCL.

- **`BasedOnPropertiesOfContextClass`**. Esta clase es adecuada exclusivamente para modelar la descripción de incumplimientos de restricciones especificadas sobre un conjunto de propiedades (atributos y referencias) de la clase contexto.

- **`BasedOnIncompatibility`**. Esta clase es adecuada exclusivamente para modelar la descripción de incumplimientos de restricciones consistentes en establecer incompatibilidades entre subclases de dos clases (típicamente abstractas) del metamodelo de dominio que se hallan relacionadas con la clase contexto a través de sendas cadenas de

asociaciones. Se puede dar el caso particular de que una de las clases coincida con la propia clase contexto, en cuyo caso la cadena de asociaciones correspondiente sería nula.

- **`BasedOnContainmentRelation`**. Esta clase es adecuada exclusivamente para modelar la descripción de incumplimientos de restricciones consistentes en establecer una relación de contención entre dos conjuntos de instancias de una clase. Cada conjunto viene indicado por el extremo final de sendas cadenas de asociaciones que parten de la clase contexto y la enlazan con la clase que tipa a los elementos de los conjuntos.
- **`ScopeBased`**. Esta clase es adecuada exclusivamente para modelar la descripción de incumplimientos de restricciones cuya satisfacción depende no sólo del estado de un elemento del modelo, sino también del ámbito o población de elementos del modelo dentro del cual se halla el primero.
- **`BasedOnAssociationChain`**: Esta clase es adecuada exclusivamente para modelar la descripción de incumplimientos de restricciones sobre las estructuras de los elementos del modelo en base a cadenas de asociaciones. Por ejemplo restricciones de estructuras cíclicas, abiertas, en árbol, multiplicidad de los elementos en las cadenas, etc.

En la subsección 3.2.6 se verán ejemplos de aplicación sobre MAST-2.

3.2.3 Herramienta de verificación basada en transformación M2M

La opción de representar el resultado de verificar un modelo en forma de otro modelo conduce de forma natural a contemplar el proceso de verificación como una transformación M2M definida entre el metamodelo del modelo que se verifica y el metamodelo respecto al cual el modelo-resultado haya de ser conforme (en este caso el metamodelo CVD). Así, dicha transformación, para la cual se acuña la denominación de *transformación de chequeo*, aplicada sobre un modelo que se desee diagnosticar da como resultado el correspondiente modelo descriptivo de incumplimientos de restricciones (modelo CVD). La Figura 3.17 muestra la composición de una herramienta de verificación de modelos en base a la estrategia expuesta.

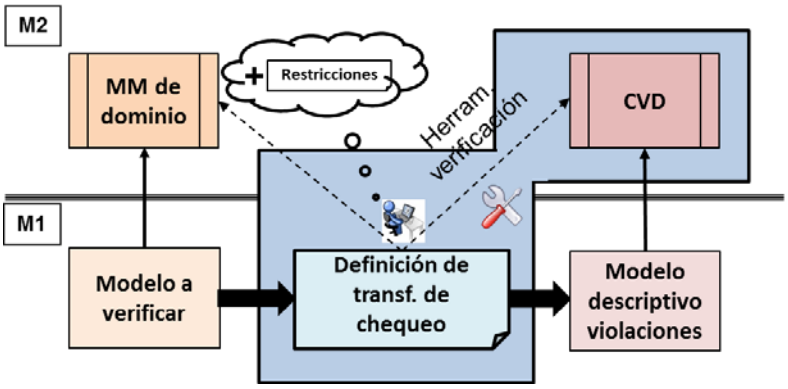


Figura 3.17 – Composición de herramienta de verificación de modelos

En la definición de una transformación de chequeo se requiere, como en toda transformación M2M, tener visibilidad (líneas punteadas en Figura 3.17) sobre los metamodelos de origen (metamodelo de dominio) y de destino (metamodelo CVD). Sin embargo, no es necesario incluir ni adjuntar explícitamente las restricciones, sino que su lógica aparecerá embebida dentro del

código de la transformación, por lo que basta con que el desarrollador M2M las conozca, aunque sea a partir de una fuente meramente documental.

El aspecto esencial en el diseño de una transformación M2M de chequeo consiste en escoger, para cada restricción especificada sobre el metamodelo de dominio, una clase concreta del metamodelo CVD que sea adecuada para describir de la manera más completa posible un hipotético incumplimiento de tal restricción.

En la siguiente subsección se expone la estructura genérica de una transformación de chequeo, empleando ATL como lenguaje de transformación de modelos.

3.2.4 Implementación ATL de una transformación de chequeo

Para implementar en ATL una transformación de chequeo se ha escogido un estilo basado esencialmente en *helpers* y *called rules*. El código ATL resultante tiene una estructura interna muy regular, siguiendo un patrón uniforme y fácilmente automatizable, lo que puede ser aprovechado favorablemente para desarrollar cada herramienta de verificación. Dicha estructura puede comprobarse en el esqueleto de código ATL mostrado en el Código 3.1, el cual puede descomponerse en cuatro bloques, sombreados mediante distintos colores.

```
module DomainMM_2_CVD;
create OUTmodel : CVD from INmodel : DomainMM;
-- ************************ ATTRIBUTES

-- ----------------
-- DomainMM classes (for each Class_J, J = 1, 2, ..., N)
-- ----------------
helper def : Class_J : Ecore!EClass = DomainMM!Class_J;

-- --------------------
-- DomainMM attributes (for each ClassJ, J = 1, 2, ..., N)
-- --------------------
helper def : Class_J_Attr_1 : Ecore!EAttribute =
    thisModule.Class_J.eAttributes -> select(a | a.name = 'Attr_1') -> first();
...
helper def : Class_J_Attr_M : Ecore!EAttribute =
    thisModule.Class_J.eAttributes -> select(a | a.name = 'Attr_M') -> first();

-- --------------------
-- DomainMM references (for each ClassJ, J = 1, 2, ..., N)
-- --------------------
helper def : Class_J_Ref_1 : Ecore!EReference =
    thisModule.Class_J.eReferences -> select(r | r.name = 'Ref_1') -> first();
...
helper def : Class_J_Ref_P : Ecore!EReference =
    thisModule.Class_J.eReferences -> select(r | r.name = 'Ref_P') -> first();

-- ************************ HELPERS

-- Helpers corresponding to the constraints specified for DomainMM

-- For each ClassJ, J = 1, 2, ..., N / ClassJ is context of constraints and
-- for each constraint_JK specified for ClassJ, K = 1, 2, ..., Q

helper context DomainMM!Class_J def : constraint_JK() : Boolean =
   constraint_JK_OCLexpr;

-- ************************ CALLED RULES

--- ------------------------------------------------------------
--- Entrypoint rule of this ATL transfo. It creates a CVD model.
--- ------------------------------------------------------------
```

```
entrypoint rule Create_CVDmodel() {
   using {
      -- For each ClassJ, J = 1, 2, ..., N / ClassJ is context of constraints
      ClassJ_Seq : Sequence(DomainMM!ClassJ) = DomainMM!ClassJ.allInstances();
   }
   to
      output : CVD!CVD_Model (
--       descriptions <- {do section}
   )
   do {
      -- For each ClassJ, J = 1, 2, ..., N / ClassJ is context of constraints

      -- --------------------------------
      -- Constraints checking for Class_J
      -- --------------------------------
      for (e in Class_J_Seq) {

         -- For each constraint_JK specified for ClassJ, K = 1, 2, ..., Q

         -- Call to the corresponding ckecker helper
          if (not e.constraint_JK())
            thisModule.CalledRuleName(output,
                                      'constraint_JK',
                                      'The constraint is intended to...',
                                       #Severity,
                                       e,
                                       ...);
      }
   }
}

-- ------------------------------------------------------------------
-- Rules for creation of CVD instances.   Rule name = CVD class name
-- ------------------------------------------------------------------

rule CVD (...) {
       ...
}

rule BasedOnPropertiesOfContextClass (...) {
       ...
}

rule BasedOnContainmentRelation (...) {
       ...
}

rule ScopeBased (...) {...}
```

Código 3.1 – Estructura de una transformación de chequeo

A continuación se describe cada bloque:

- **Bloque I (*sección de atributos*)**. Contiene la declaración de atributos (*helper attributes*) correspondientes a las clases y propiedades de clases en el metamodelo de dominio relacionadas directa o indirectamente con la especificación de restricciones sobre él.

  Por tanto, el conjunto de restricciones planteado define completamente este bloque.

- **Bloque II (*sección de helpers*)**. Contiene la definición de *helpers* funcionales (uno por cada restricción impuesta sobre el metamodelo de dominio) que permiten consultar a los elementos del modelo a verificar si cumplen o no las restricciones especificadas para su tipo (clase). Por tanto, retornan un booleano y cada uno declara como contexto la clase contexto de la restricción correspondiente.

Así pues, el conjunto de restricciones planteado también define completamente este segundo bloque, siendo su código análogo para diferentes transformaciones de chequeo definidas sobre metamodelos de dominio diferentes o sobre distintos paquetes de restricciones especificados sobre un mismo metamodelo.

Una buena práctica es agrupar la definición de *helpers* según su contexto de aplicación; así, primero se tienen los correspondientes a la clase ContextClass1, a continuación los correspondientes a la clase ContextClass2, etc., habiendo en cada grupo tantos *helpers* como restricciones impuestas sobre la respectiva clase.

Es importante notar que, gracias a la alineación de ATL y OCL, el cuerpo de un *helper* coincide exactamente con la expresión OCL (booleana) con que se formularía la restricción para el metamodelo de dominio. Esto será aprovechado posteriormente para automatizar la generación del código.

- **Bloque III (*regla de entrada*)**. Es una *called rule* especial, que se invoca en primer lugar al ejecutar la transformación y genera el contenedor principal del modelo de salida, esto es, la instancia `CVD_Model` que ha de albergar las instancias de descripción de los incumplimientos detectados.

  En la sección *using* de esta regla se declaran variables locales de tipo secuencia para cada una de las clases contexto de restricciones, cada una destinada a contener todas las instancias de la correspondiente clase existentes en el modelo de entrada. Estas secuencias serán recorridas en la parte imperativa de la regla para chequear en cada elemento de modelo el cumplimiento de las restricciones pertinentes.

  El código de la parte imperativa sigue también una estructura muy concreta, con bucles para recorrer cada secuencia de elementos de modelo y sobre cada uno consultando, gracias al *helper* al efecto, si NO cumple cada una de las restricciones que le son aplicables, en cuyo caso se genera en el modelo de salida una instancia de descripción de incumplimiento. Para ello se invoca a la *called rule* apropiada de entre las definidas en el bloque siguiente.

  Por tanto, dada esta estructura tan bien determinada, el código de este bloque es análogo para toda transformación definida sobre cualquier par metamodelo + restricciones.

- **Bloque IV (*sección de called rules*)**. Es un bloque fijo, idéntico en cualquier transformación de chequeo (siempre y cuando el metamodelo destino sea el mismo), y consta de la definición de una *called rule* por cada clase concreta en dicho metamodelo destino. Cada una de estas reglas genera una instancia de tal clase concreta de descripción de incumplimiento y la incorpora al modelo de salida. Se ha optado por asignar a cada regla el nombre de la clase correspondiente.

  Todas aceptan como parámetros el contenedor principal del modelo de salida al cual se incorporará la instancia generada, así como el nombre y descripción de la restricción incumplida junto al elemento del modelo donde se ha localizado el incumplimiento (elemento contextual) y la severidad que se quiere atribuir a tal violación. Además, cada regla acepta otros parámetros específicos de cada una, acordes a las propiedades de la clase concreta de descripción de incumplimiento a que dan lugar. Es en los *bindings* de estas reglas donde se inicializan dichas propiedades.

Por tanto, el código ATL de una transformación de chequeo tiene una estructura uniforme, con una parte variable (secciones de atributos, *helpers* y regla de entrada) que depende del metamodelo fuente y las restricciones, pero con un patrón estructural que puede ser fácilmente adaptado y una parte fija (sección de *called rules*), siempre y cuando se mantenga inalterado el metamodelo de destino.

3.2.5 Automatización de la estrategia diseñada

La estrategia diseñada para la verificación de modelos requiere desarrollar una herramienta diferente (implementar una trasformación de chequeo diferente) para cada pareja metamodelo de dominio + conjunto de restricciones sobre él impuestas, lo cual contrasta con el objetivo planteado de desarrollar una herramienta genérica que pueda aplicarse para verificar modelos independientemente de su metamodelo y de las restricciones que se apliquen. La Figura 3.18 y la Figura 3.19 representan ambos escenarios.

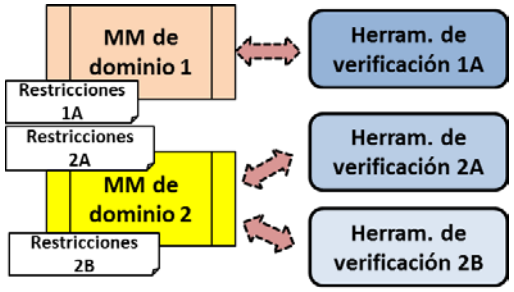


Figura 3.18 – Herramientas específicas de verificación

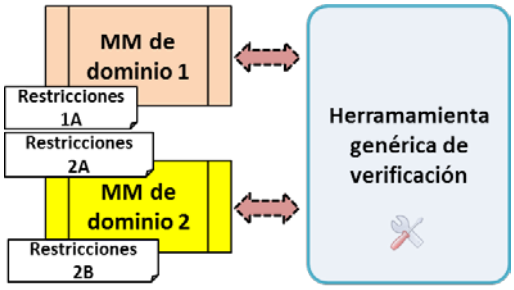


Figura 3.19 – Herramienta genérica de verificación

Puesto que no es asumible tratar de abstraer los infinitos metamodelos de dominio que el enfoque DSM promueve y puesto que se ha diseñado una estrategia que posibilita el desarrollo sistemático de herramientas específicas de verificación, la herramienta genérica que se propone está basada en construir bajo demanda la herramienta (transformación) de verificación correspondiente a cada caso, tal y como muestra la Figura 3.20.

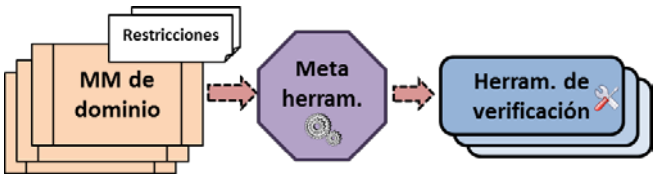


Figura 3.20 – Metaherramienta generadora de herramientas específicas

Dado que la metaherramienta ha de producir transformaciones M2M, su fundamento funcional es una HOT, la cual se aborda posteriormente dentro de esta misma subsección.

Para generar cada transformación de chequeo, la metaherramienta debe disponer como entrada de toda la información que un desarrollador manejaría para su implementación manual. Esta información consiste en el propio conjunto de restricciones junto con la forma elegida de modelar los datos para la descripción de incumplimientos de cada una de ellas, en este caso según el metamodelo CVD, esto es, el mapeo de cada restricción a una clase CVD. Más concretamente, se requiere información, no sólo sobre qué tipo de descripción de incumplimiento se asigna a una restricción, sino también aquella relativa a qué elementos del metamodelo de dominio (normalmente atributos, asociaciones o cadenas de asociaciones) se asignan a las propiedades de las instancias CVD de descripción de violaciones.

Toda esta información vinculada a una restricción (datos propios, como nombre, expresión OCL y clase contexto así como datos de mapeo) y requerida para desarrollar una transformación de chequeo, constituye la *caracterización de una restricción*. Así pues, una metaherramienta capaz de generar cada herramienta específica de verificación necesita como entrada un modelo que represente la información constituida por el conjunto de caracterizaciones correspondientes a las restricciones especificadas. La Figura 3.21 muestra el escenario considerado.

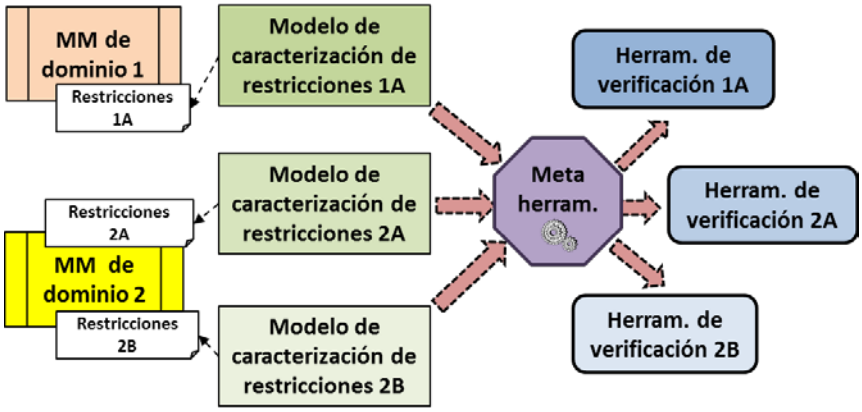


Figura 3.21 – Modelos de caracterización de restricciones

Para formalizar la estructura de estos modelos, se propone un metamodelo de *Caracterización de Restricciones* (CC). En la Figura 3.22 se muestra su rol en conjunción con los metamodelos de dominio y CVD. El metamodelo CC será objeto de un apartado posterior. A continuación se profundiza en el núcleo de la metaherramienta desarrollada: la HOT encargada de generar transformaciones de chequeo específicas.

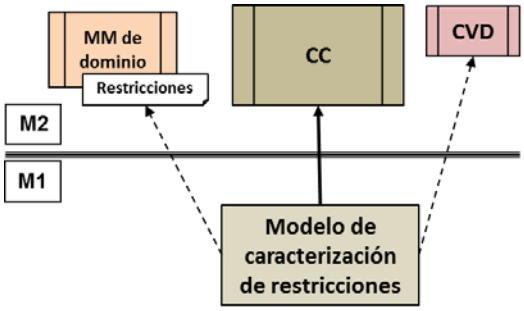


Figura 3.22 – Relación entre un par metamodelo + restricciones y el metamodelo CVD por medio de un modelo CC

Núcleo HOT de la metaherramienta

Se trata de una *HOT de síntesis*, que, como se muestra en la Figura 3.23, acepta como entrada un modelo conforme al metamodelo CC y produce como salida un modelo de transformación conforme al metamodelo del lenguaje de transformación de modelos empleado (ATL). Este modelo representa la transformación de chequeo correspondiente al par metamodelo de dominio + restricciones, junto a las decisiones tomadas respecto a cómo modelar los posibles incumplimientos de éstas últimas.

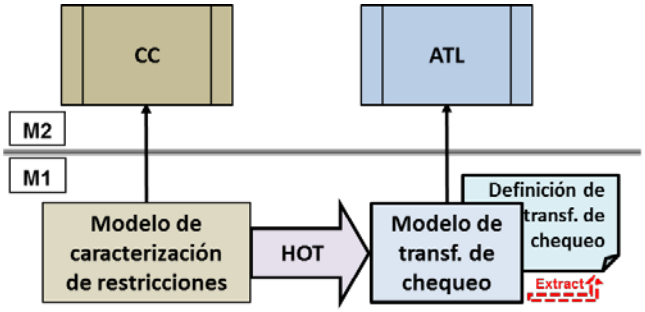


Figura 3.23 – HOT generadora de transformaciones de chequeo

Para obtener la transformación de chequeo propiamente dicha, codificada en la sintaxis textual concreta de ATL, sólo restaría la serialización (extracción) del modelo producido.

Metamodelo CC

Este metamodelo constituye una propuesta de metamodelo para los modelos mediante los cuales se formula la información de caracterización de las restricciones especificadas sobre un metamodelo de dominio.

El metamodelo CC presenta una estructura convencional, con una clase contenedor principal (`CC_Model`) y una clase raíz (`ConstraintCharacterization`) de la que heredan el resto de clases, conformando en conjunto la jerarquía de clases destinadas al modelado de (los datos necesarios para caracterizar) restricciones. La Figura 3.24 muestra las clases contenedor principal y raíz, así como las principales subclases de ésta última.

Un modelo conforme a CC posee una única instancia de `CC_Model`, la cual contiene a través de la asociación `constraintCharacterizations` al resto de elementos de modelo, instancias de `ConstraintCharacterization` o de cualquiera de sus subclases. A continuación se describen brevemente la clase raíz `ConstraintCharacterization` y sus subclases de primer nivel:

- **`ConstraintCharacterization`**. Esta clase permite modelar de forma mínima la caracterización de restricciones de cualquier tipo.

  Únicamente con esta clase raíz sería suficiente para formular como modelo el conjunto de caracterizaciones correspondientes a las restricciones especificadas sobre un metamodelo, pues la información que describe es apta para cualquier restricción, independientemente de su naturaleza, semántica o formulación OCL.

- **`BasedOnPropertiesOfContextClass`**. Esta clase es adecuada exclusivamente para modelar la caracterización de restricciones especificadas sobre un conjunto de propiedades (atributos y referencias) de la clase contexto.

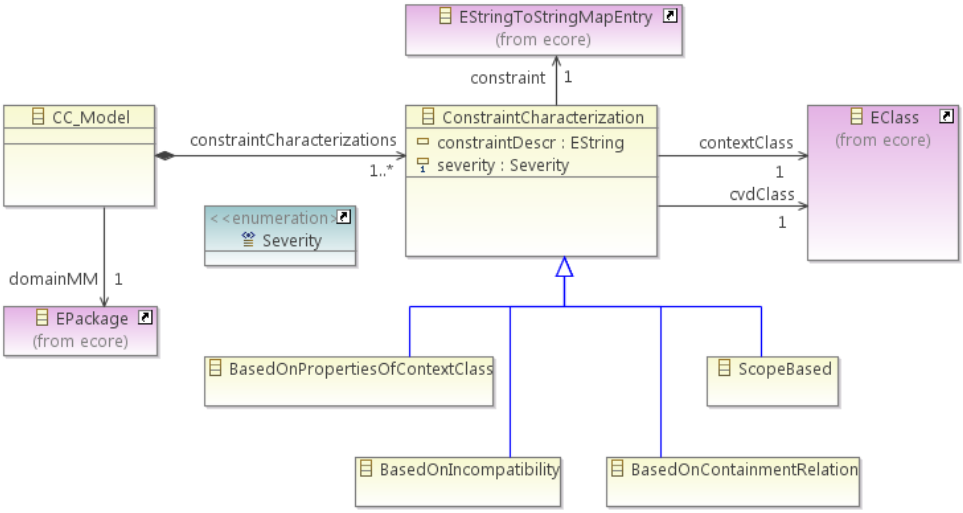


Figura 3.24 – Núcleo del metamodelo CC

- **`BasedOnIncompatibility`**. Esta clase es adecuada exclusivamente para modelar la caracterización de restricciones consistentes en establecer incompatibilidades entre subclases de dos clases (típicamente abstractas) del metamodelo de dominio que se hallan relacionadas con la clase contexto a través de sendas cadenas de asociaciones.

  Se puede dar el caso particular de que una de las clases coincida con la propia clase contexto, en cuyo caso la correspondiente cadena de asociaciones sería nula.

- **`BasedOnContainmentRelation`**. Esta clase es adecuada exclusivamente para modelar la caracterización de restricciones consistentes en establecer una relación de contención entre dos conjuntos de instancias de una clase. Cada conjunto viene indicado por el extremo final de sendas cadenas de asociaciones que parten de la clase contexto y la enlazan con la clase que tipa a los elementos de los conjuntos.

- **`ScopeBased`**. Esta clase es adecuada exclusivamente para modelar la caracterización de restricciones cuya satisfacción depende no sólo del estado de un elemento de modelo, sino también del ámbito o población de elementos de modelo dentro del cual el primero se halla.

- **`BasedOnAssociationChain`**: Esta clase es adecuada exclusivamente para modelar la caracterización de restricciones sobre las estructuras de los elementos del modelo en base a cadenas de asociaciones. Por ejemplo restricciones de estructuras cíclicas, abiertas, en árbol, multiplicidad de los elementos en las cadenas, etc.

En la siguiente subsección se muestran ejemplos de aplicación sobre MAST-2.

3.2.6 Aplicación de la metaherramienta en MAST-2

Tal y como se expuso en la subsección 2.3.3, el dominio MAST-2 se encuentra formalizado mediante un metamodelo con un importante grado de laxitud y en consecuencia, está complementado con un conjunto de restricciones de integridad, denominadas *Restricciones MAST-2*. Además, las diversas herramientas del entorno MAST-2 definen paquetes adicionales de restricciones específicas. En todos los casos, las restricciones se han formulado en OCL. Por tanto, este entorno representa un banco de trabajo ideal sobre el que aplicar la estrategia diseñada para verificación de modelos.

Para dichos conjuntos de restricciones, tanto de integridad como específicas de herramientas, se han formulado los correspondientes modelos caracterizadores (conformes al metamodelo CC) y a partir de ellos se han generado las respectivas transformaciones de chequeo. A efectos ilustrativos, esta subsección se centra en el paquete de restricciones de integridad, cuyo modelo caracterizador es `Mast2_integrity.cc.xmi` y la correspondiente transformación de chequeo es `Mast2_integrity_to_CVD.atl`. Como restricciones representativas, se retoman las mismas expuestas en la subsección 2.3.3. Para cada una se presenta, a modo de diagrama de objetos, la formulación de su caracterización en el modelo caracterizador y los fragmentos de código ATL encargados de chequear su cumplimiento (*helper*) y de, en caso de incumplimiento, generar en el modelo CVD de salida la correspondiente instancia descriptiva de violación. Como puede observarse en el código de los *helpers*, su cuerpo coincide con la expresión OCL de cada restricción.

- Invariante ***i_1_1_a***. La Figura 3.25 muestra la formulación de su caracterización.

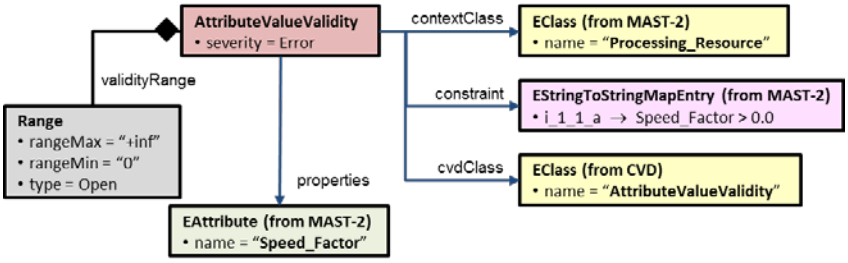


Figura 3.25 – Formulación de la caracterización del invariante *i_1_1_a*

```
helper context Mast2!Processing_Resource def : i_1_1_a() : Boolean =
   self.Speed_Factor > 0.0;
```

Código 3.2 – *Helper* correspondiente al invariante *i_1_1_a*

```
for (pr in Processing_Resource_Seq) {
   if (not pr.i_1_1_a())
      thisModule.AttributeValueValidity(output,
                                        'i_1_1_a',
                                        'This constraint...',
                                        #Error,
                                        pr,
                                        thisModule.Processing_Resource_Speed_Factor,
                                        '0.0',
                                        '+inf',
                                        #Open);
```

Código 3.3 – Chequeo del cumplimiento del invariante *i_1_1_a*

A modo ilustrativo, si se tiene un modelo MAST-2 en donde existe un objeto de la clase `Regular_Processor` (subclase concreta de *`Processing_Resource`*), llamado Proc1, que incumple esta restricción, se genera una instancia de `AttributeValueValidity` en el modelo CVD de salida. La Figura 3.26 muestra los elementos de modelo generados y referenciados.

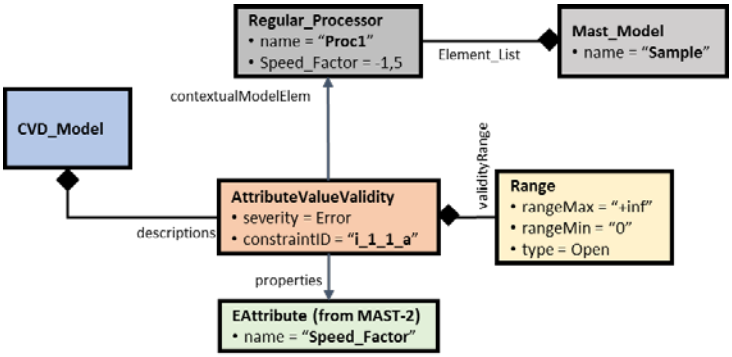


Figura 3.26 – Incumplimiento del invariante *i_1_1_a*

- Invariante ***i_2_1_a***. La Figura 3.27 muestra la formulación de su caracterización.

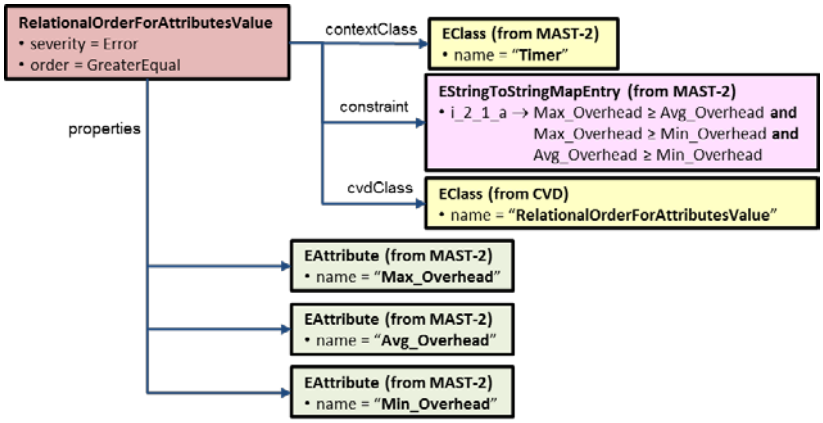


Figura 3.27 – Formulación de la caracterización del invariante *i_2_1_a*

```
helper context Mast2!Timer def : i_2_1_a() : Boolean =
   self.Max_Overhead >= self.Avg_Overhead and
   self.Max_Overhead >= self.Min_Overhead and
   self.Avg_Overhead >= self.Min_Overhead;
```

Código 3.4 – *Helper* correspondiente al invariante *i_2_1_a*

```
for (t in Timer_Seq) {
   if (not t.i_2_1_a())
      thisModule.RelationalOrderForAttributesValue(output,
                                          'i_2_1_a',
                                          'This constraint...',
                                          #Error,
                                          t,
                                          #GreaterEqual,
                                          Sequence{thisModule.Timer_Max_Overhead,
                                                   thisModule.Timer_Avg_Overhead,
                                                   thisModule.Timer_Min_Overhead});
```

Código 3.5 – Chequeo del cumplimiento del invariante *i_2_1_a*

A modo ilustrativo, si se tiene un modelo MAST-2 en donde existe un objeto de la clase `Ticker` (subclase concreta de *`Timer`*), llamado Timer1, que incumple esta restricción, se genera una instancia de `RelationalOrderForAttributesValue` en el modelo CVD de salida. La Figura 3.28 muestra los elementos de modelo generados y referenciados.

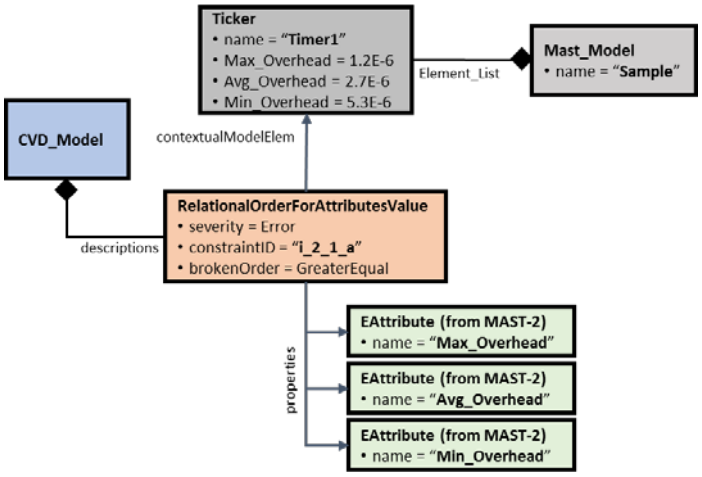


Figura 3.28 – Incumplimiento del invariante *i_2_1_a*

- Invariante ***i_3_5_a_I***. La Figura 3.29 muestra la formulación de su caracterización.

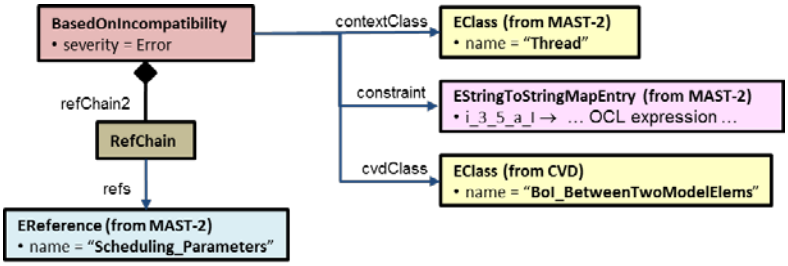


Figura 3.29 – Formulación de la caracterización del invariante *i_3_5_a_I*

```
helper context Mast2!Thread def : i_3_5_a_I() : Boolean =
   self.Scheduling_Parameters.oclIsKindOf(Mast2!Interrupt_Based_Params) or
   self.Scheduling_Parameters.oclIsTypeOf(Mast2!Non_Preemptible_FP_Params) or
   self.Scheduling_Parameters.oclIsTypeOf(Mast2!Fixed_Priority_Params) or
   self.Scheduling_Parameters.oclIsTypeOf(Mast2!Polling_Params) or
   self.Scheduling_Parameters.oclIsKindOf(Mast2!Periodic_Server_Params) or
   self.Scheduling_Parameters.oclIsKindOf(Mast2!EDF_Based_Params) or
   self.Scheduling_Parameters.oclIsKindOf(Mast2!Timetable_Driven_Params);
```

Código 3.6 – *Helper* correspondiente al invariante *i_3_5_a_I*

```
for (th in Thread_Seq) {
   if (not th.i_3_5_a_I())
      thisModule.ReferenceTypeValidity(output,
                           'i_3_5_a_I',
                           'This constraint...',
                           #Error,
                           th,
                           thisModule.Schedulable_Resource_Scheduling_Parameters,
                           Sequence{thisModule.Priority_Based_Params,
                                    thisModule.Interrupt_Based_Params,
                                    thisModule.EDF_Based_Params,
                                    thisModule.Timetable_Driven_Params});
```

Código 3.7 – Chequeo del cumplimiento del invariante *i_3_5_a_I*

A modo ilustrativo, si se tiene un modelo MAST-2 en donde existe un objeto de la clase `Thread`, llamado Thread1, que incumple esta restricción, se genera una instancia de `BoI_BetweenTwoModelElems` en el modelo CVD de salida. La Figura 3.30 muestra los elementos de modelo generados y referenciados.

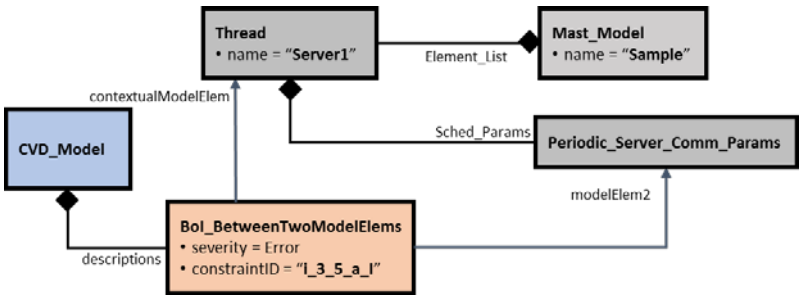


Figura 3.30 – Incumplimiento del invariante *i_3_5_a_I*

- Invariante ***i_3_5_a_II***. La Figura 3.31 muestra la formulación de su caracterización.

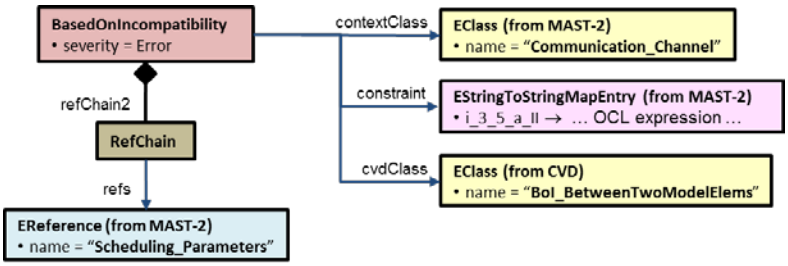


Figura 3.31 – Formulación de la caracterización del invariante *i_3_5_a_II*

```
helper context Mast2!Communication_Channel def : i_3_5_a_II() : Boolean =
   self.Scheduling_Parameters.oclIsTypeOf(Mast2!Non_Preemptible_FP_Params) or
   self.Scheduling_Parameters.oclIsTypeOf(Mast2!Fixed_Priority_Params) or
   self.Scheduling_Parameters.oclIsTypeOf(Mast2!Polling_Params) or
   self.Scheduling_Parameters.oclIsKindOf(Mast2!Periodic_Server_Comm_Params) or
   self.Scheduling_Parameters.oclIsKindOf(Mast2!EDF_Based_Params) or
   self.Scheduling_Parameters.oclIsKindOf(Mast2!Timetable_Driven_Params) or
   self.Scheduling_Parameters.oclIsTypeOf(Mast2!AFDX_Virtual_Link);
```

Código 3.8 – *Helper* correspondiente al invariante *i_3_5_a_II*

```
for (c in Communication_Channel_Seq) {
   if (not c.i_3_5_a_II())
      thisModule.ReferenceTypeValidity(output,
                           'i_3_5_a_II',
                           'This constraint...',
                           #Error,
                           th,
                           thisModule.Schedulable_Resource_Scheduling_Parameters,
                           Sequence{thisModule.Priority_Based_Params,
                                    thisModule.EDF_Based_Params,
                                    thisModule.Timetable_Driven_Comm_Params,
                                    thisModule.AFDX_Virtual_Link});
```

Código 3.9 – Chequeo del cumplimiento del invariante *i_3_5_a_II*

- Invariante ***i_4_5_a***. La Figura 3.32 muestra la formulación de su caracterización.

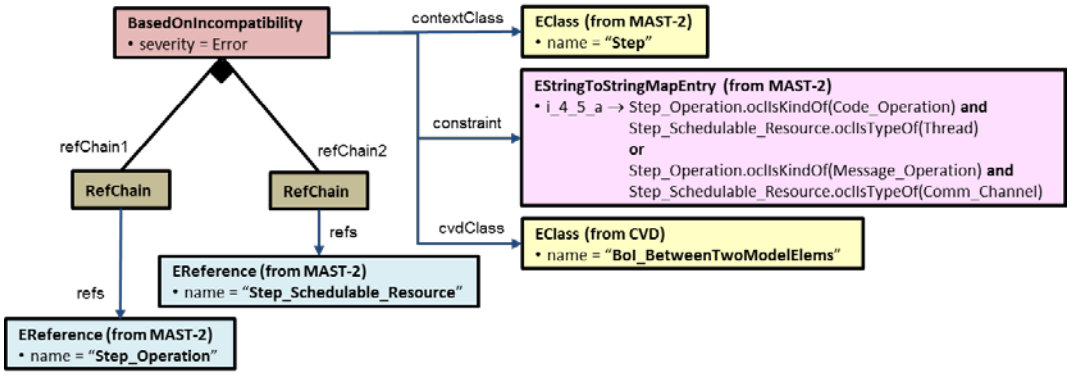


Figura 3.32 – Formulación de la caracterización del invariante *i_4_5_a*

```
helper context Mast2!Step def : i_4_5_a() : Boolean =
   (self.Step_Operation.oclIsKindOf(Mast2!Code_Operation) and
    self.Step_Schedulable_Resource.oclIsTypeOf(Mast2!Thread)
   )
   or
   (self.Step_Operation.oclIsKindOf(Mast2!Message_Operation) and
    self.Step_Schedulable_Resource.oclIsTypeOf(Mast2!Communication_Channel)
   );
```

Código 3.10 – *Helper* correspondiente al invariante *i_4_5_a*

```
for (st in Step_Seq) {
   if (not st.i_4_5_a())
      thisModule.BoI_BetweenTwoModelElems(output,
                                          'i_4_5_a',
                                          'This constraint...',
                                          #Error,
                                          st,
                                          st.Step_Operation,
                                          st.Step_Schedulable_Resource);
```

Código 3.11 – Chequeo del cumplimiento del invariante *i_4_5_a*

A modo ilustrativo, si se tiene un modelo MAST-2 en donde existe un objeto de la clase `Step` que incumple esta restricción, se genera una instancia de `BoI_BetweenTwoModelElems` en el modelo CVD de salida. La Figura 3.33 muestra los elementos de modelo generados y referenciados.

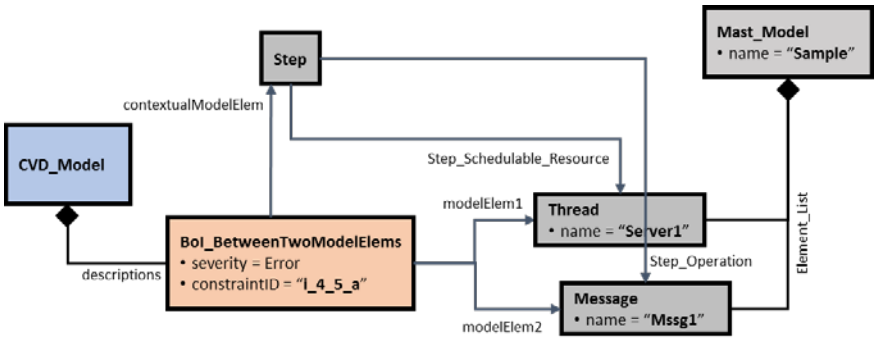


Figura 3.33 – Incumplimiento del invariante *i_4_5_a*

- Invariante ***i_4_1_a***. La Figura 3.34 muestra la formulación de su caracterización.

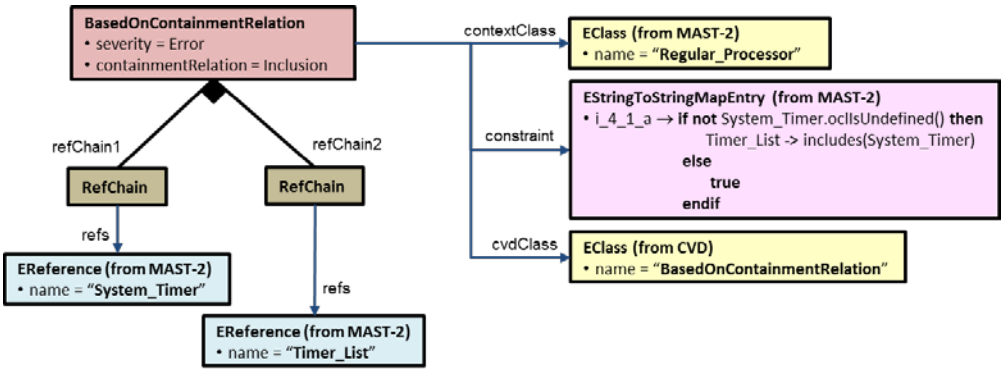


Figura 3.34 – Formulación de la caracterización del invariante *i_4_1_a*

```
helper context Mast2!Regular_Processor def : i_4_1_a() : Boolean =
   if not self.System_Timer.oclIsUndefined() then
      self.Timer_List -> includes(self.System_Timer)
   else
      true
   endif;
```

Código 3.12 – *Helper* correspondiente al invariante *i_4_1_a*

```
for (rp in Regular_Processor_Seq) {
   if (not rp.i_4_1_a())
      thisModule.BasedOnContainmentRelation(output,
                                            'i_4_1_a',
                                            'This constraint…',
                                            #Error,
                                            rp,
                                            #Inclusion,
                                            rp.System_Timer,
                                            rp.Timer_List);
```

Código 3.13 – Chequeo del cumplimiento del invariante *i_4_1_a*

A modo ilustrativo, si se tiene un modelo MAST-2 en donde existe un objeto de la clase `Regular_Processor`, llamado Proc1, que incumple esta restricción, se genera una instancia de `BasedOnContainmentRelation` en el modelo CVD de salida. La Figura 3.35 muestra los elementos de modelo generados y referenciados.

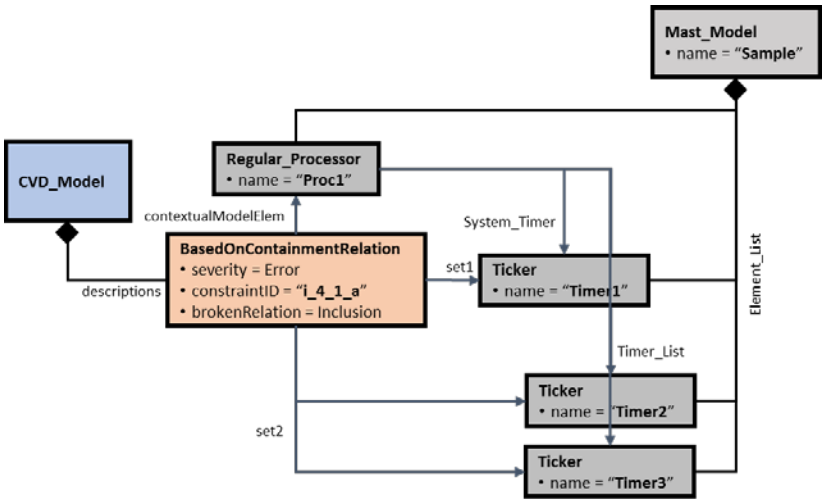


Figura 3.35 – Incumplimiento del invariante *i_4_1_a*

3.2.7 Enfoques relacionados

Aplicar un enfoque M2M para abordar el problema de la verificación no es algo nuevo, pues ya ha sido esbozado en [128] y aplicado en trabajos posteriores, como [129] y [130]. En el trabajo seminal de Bézivin, la aplicación de una transformación M2M sobre el modelo a verificar da como resultado lo que se denomina un *modelo de diagnóstico*, conforme a un metamodelo llamado *Problems*. Se trata de un metamodelo extremadamente simple, con una única clase que define tres atributos: severidad, localización y descripción. La propuesta aquí presentada extiende esta idea nuclear, desarrollando una estrategia más ambiciosa en base a un metamodelo mucho más elaborado (CVD). Además, los autores sólo esbozan un patrón para implementar manualmente la transformación correspondiente a cada formalización de dominio. Por su parte, Diguet propone un metamodelo de diagnosis llamado VERIF y usa ATL para implementar una transformación M2M para chequeo de restricciones de corrección sintáctica sobre modelos MARTE, como paso preliminar en su transformación principal de MARTE a AADL. Sin embargo, aun siendo más elaborado que el metamodelo *Problems* de Bézivin, el metamodelo VERIF es aún bastante simple y, de nuevo, el trabajo se centra sólo en una transformación específica para un caso concreto, aunque puede ser tomada como una plantilla. Nuestra propuesta sobrepasa el propósito de estos trabajos, aspirando a proporcionar una solución genérica independiente de la formalización del dominio. Este carácter genérico también es reclamado por el tercer trabajo mencionado, el cual aborda la detección de problemas en modelos a través de transformaciones QVTr desde modelos de entrada conformes a cualquier metamodelo basado en MOF a modelos resultado, conformes al metamodelo *pResults,* donde las ocurrencias de problemas son reportadas de una manera estructurada y concisa.

Un trabajo relativamente cercano, aunque siguiendo un enfoque diferente es [131]. Los autores proponen un método para chequear eficientemente restricciones OCL por medio de SQL. La idea nuclear consiste en reducir el problema a chequear si consultas SQL arrojan un resultado vacío. Dada una restricción OCL, es posible construir una consulta SQL que retorne todas las instancias que la violan. Por tanto, la restricción OCL es satisfecha si y sólo si su correspondiente consulta SQL retorna el conjunto vacío. La computación de tales consultas es llevada a cabo de forma incremental por un DBMS relacional.

Es importante remarcar que el problema abordado, esto es, la verificación de la satisfacción de invariantes, no trata la validación de la propia formalización del dominio (metamodelo + restricciones). En este sentido, al considerar un conjunto de invariantes especificados sobre un metamodelo de dominio, se supone que tal conjunto es perfectamente válido, satisfaciendo las típicas propiedades de corrección: corrección sintáctica, ni sobrerrestricción ni infrarrestricción de metamodelo, consistencia, independencia, posibilidad de satisfacción, no subsunción, no redundancia, etc. En [132] puede encontrarse una distinción clara entre verificación de modelos terminales y validación de la formalización de dominio. De hecho, existe una importante cantidad de investigación publicada sobre el tema de la validación, como por ejemplo [133-135]. Sin embargo, esta dimensión queda fuera del ámbito de esta Tesis.

3.3 Herramienta para Construcción de Modelos Acordes a Vistas de Dominio

La formalización de dominios de aplicación genéricos requiere definir metamodelos complejos con muchos detalles y opciones. En la práctica, cada desarrollador sólo utiliza aquellos aspectos del mismo que se refieren a la metodología y tecnología concreta que utiliza. Para el desarrollador que trabaja en un contexto específico, la complejidad del metamodelo de dominio le supone una dificultad innecesaria, ya que puede:

- No ser experto en la totalidad del dominio, sino sólo en algunos aspectos parciales.
- Sólo utilizar un conjunto limitado de patrones de diseño o plataformas de ejecución respecto de los contemplados en la generalidad del dominio.
- Desarrollar herramientas destinadas a modelos de sistemas con unas características restringidas dentro de la variabilidad del dominio.
- Estar interesado sólo en una vista parcial de la información del modelo.

Resolver estas dificultades descomponiendo los metamodelos de dominio complejos en otros parciales más simples que se adapten a los intereses de cada desarrollador puede no ser recomendable o incluso no estar permitido si el objetivo es dar una cobertura homogénea al modelado de sistemas con características específicas diferentes, pero de forma que a todos ellos se les puedan aplicar técnicas de procesamiento y herramientas comunes. En cambio, manteniendo como base el metamodelo de dominio original, lo que sí puede hacerse es facilitar la tarea del diseñador proporcionándole herramientas adaptadas que le ofrezcan una visión de la información orientada a los aspectos de su interés. Una opción para ello es trabajar sobre metamodelos auxiliares más sencillos que sólo contemplen tales aspectos, aunque esta estrategia ha de verse necesariamente complementada para que los modelos que se manejen sean conformes al metamodelo de dominio original, de forma que sean compatibles con herramientas legadas y asimismo sean el punto de partida hacia nuevas áreas y aspectos específicos de interés para otros diseñadores. En ciertos casos triviales, los metamodelos de soporte de estas características específicas pueden ser un mero subconjunto del metamodelo de dominio; en esos casos la conformidad respecto a él de los modelos manejados está asegurada. Sin embargo, en general, tales metamodelos específicos pueden exhibir diferencias estructurales con el original, y por tanto los modelos no ser directamente conformes a este último.

Especificación de vistas sobre un metamodelo

Una forma de facilitar al diseñador la comprensión y el uso de un dominio de conocimiento genérico formalizado mediante un metamodelo complejo es mediante la especificación formal de vistas especializadas del mismo.

> *Un modelo conforme a un metamodelo y cuya estructura satisfaga la especificación de una vista se dice que es acorde a esa vista.*

La Figura 3.36 ilustra esta idea, mostrando en la parte superior (capa M2) un metamodelo de dominio sobre el que un experto en él ha especificado dos vistas A y B que lo interpretan y en la parte inferior (capa M1) tres modelos conformes al metamodelo. Éstos pueden ser acordes a la Vista A (Modelo 1) o a la Vista B (Modelo 3) o a ninguna de ellas (como el Modelo 2, que

simplemente es conforme). Si existiese algún tipo de intersección no nula entre las vistas, nada impediría que existiesen modelos acordes a ambas.

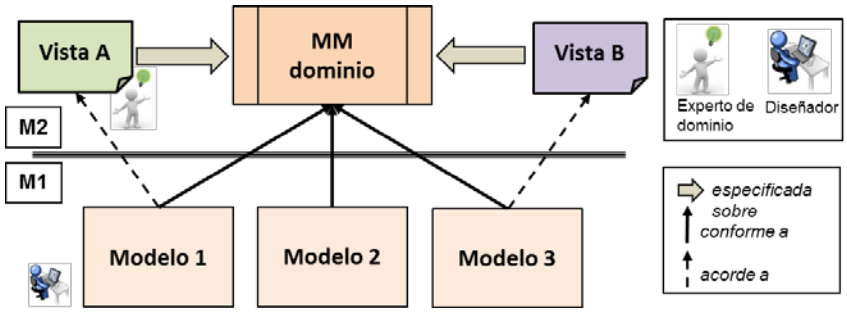


Figura 3.36 – Especificación de vistas sobre un metamodelo de dominio

Motivación y conveniencia de la herramienta genérica contribuida

En esta sección se desarrolla el concepto de vista especializada de un dominio genérico, se describe su alcance y se proponen estrategias para desarrollar herramientas que le den soporte. Dentro del objetivo de este capítulo, se aborda el diseño de una herramienta genérica para facilitar la construcción de modelos acordes a una vista especificada sobre un metamodelo.

Si la estrategia se implementase mediante herramientas MDSE estándar, sería necesario el desarrollo *ad hoc* de una herramienta específica para cada pareja de metamodelo y vista. Ésta sería una solución no deseable, pues:

- Dado un metamodelo y sus vistas asociadas, aunque éste se mantenga inalterado:
  - Las vistas pueden sufrir modificaciones en su especificación, por lo que cada herramienta tendría que ser convenientemente adaptada.
  - Nuevas vistas pueden ser especificadas, requiriendo el desarrollo de las correspondientes nuevas herramientas.
- Si además el metamodelo evoluciona, en general puede ser necesaria la reformulación de las vistas especificadas sobre él. De nuevo, esto implicaría la necesidad de rediseñar las herramientas respectivas.
- La incorporación a un entorno de un nuevo metamodelo sobre el que existen vistas especificadas requeriría la creación de nuevas herramientas de construcción.

Por todo ello, se propone una herramienta genérica cuyo objetivo es que un diseñador que trabaja sobre un dominio de conocimiento general formalizado mediante un metamodelo complejo, en base a una vista del mismo acorde a la metodología y tecnología que esté utilizando, pueda generar el modelo que necesita en dos etapas:

1. **Definición de la vista y generación de los recursos para su gestión**: Probablemente a nivel corporativo, debe definirse la vista del dominio de conocimiento que requiere la metodología y tecnología que se utiliza. Esto supone realizar los siguientes pasos:
   a. Definir la vista.
   b. Generar el metamodelo del dominio de conocimiento que corresponde a la vista.
   c. Generar la herramienta que transforma el modelo correspondiente a la vista al modelo correspondiente al dominio de conocimiento completo.

2. **Utilización de la vista para el desarrollo de los modelos del sistema que desarrolla**: El ingeniero software que desarrolla un sistema concreto debe desarrollar el modelo de su sistema que corresponde al dominio de conocimiento. Esto lo hace realizando los siguientes pasos:

    a. Define el modelo del sistema que desarrolla conforme al metamodelo de la vista.

    b. Utiliza la herramienta de transformación de modelos para transformar el modelo conforme al metamodelo de la vista en un modelo conforme al metamodelo genérico del dominio de conocimiento.

La herramienta que se desarrolla es genérica porque permite realizar el proceso sobre vistas definidas sobre cualquier metamodelo de dominio. La herramienta lleva asociado, como parte propia, el metamodelo que permite modelar las vistas para cualquier dominio de conocimiento.

En los siguientes apartados de esta sección se propone una herramienta genérica basada en una metaherramienta que bajo demanda genere los metamodelos intermedios con los que el diseñador introduce la información de acuerdo a la vista y la herramienta específica que genera el modelo final. Esto es, la metaherramienta está basada en una transformación promocionadora[55] (M2MM) y una HOT.

Herramienta basada en metaherramienta

La Figura 3.37 muestra los elementos básicos de la herramienta diseñada. La caracterización y diseño como herramienta genérica requiere la definición del ECC que describe su funcionalidad y su consiguiente formalización como metamodelo de instrucción. El ECC se tratará en una subsección posterior, aunque para facilitar la comprensión de la Figura 3.37 se indica que el *metamodelo CVS* (*Constraining Views Specification*) constituye el metamodelo de instrucción y cada modelo instancia suya describe una determinada vista.

Dada una vista especificada sobre un metamodelo, la formulación de un modelo descriptivo de tal vista instruye a la metaherramienta (basada en transformación M2MM + HOT), núcleo de la herramienta genérica de construcción, para que produzca la herramienta específica adecuada (basada en un metamodelo auxiliar + transformación M2M).

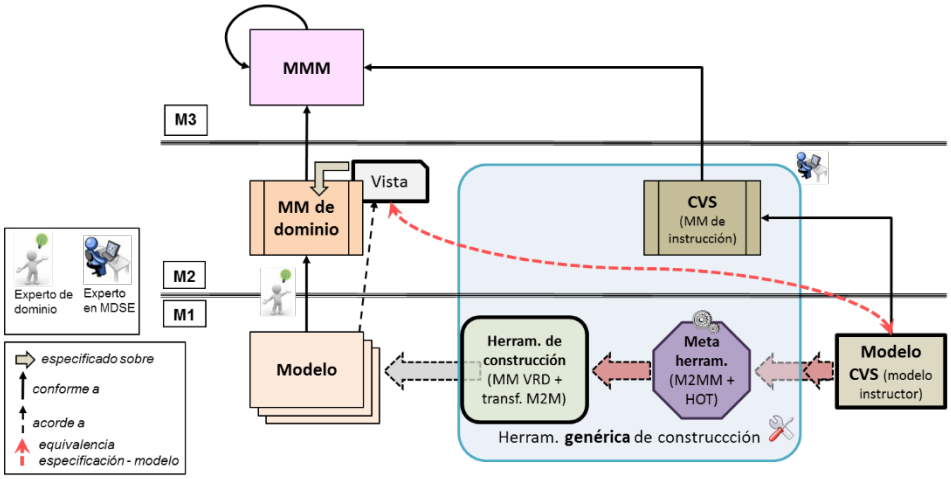


Figura 3.37 – Esquema de la herramienta genérica de construcción

[55] Una transformación que toma como entrada un artefacto en la capa M1 (modelo) y produce como salida un artefacto en la capa M2 (metamodelo).

La Figura 3.38 incorpora nuevos detalles a la Figura 3.37 a fin de resaltar el carácter universal de la herramienta diseñada. Dada cualquier vista[56] especificada sobre cualquier metamodelo, la formulación del correspondiente modelo instructor permite la obtención de la correspondiente herramienta específica.

Así, la actuación ante cualquier supuesto dentro de la casuística evolutiva enumerada anteriormente se reduce a la formulación del modelo instructor que caracteriza a la nueva (renovada o inédita) vista. A partir de este modelo se genera la nueva herramienta específica de construcción, sin necesidad de adaptación de la que ha quedado obsoleta, la cual puede desecharse.

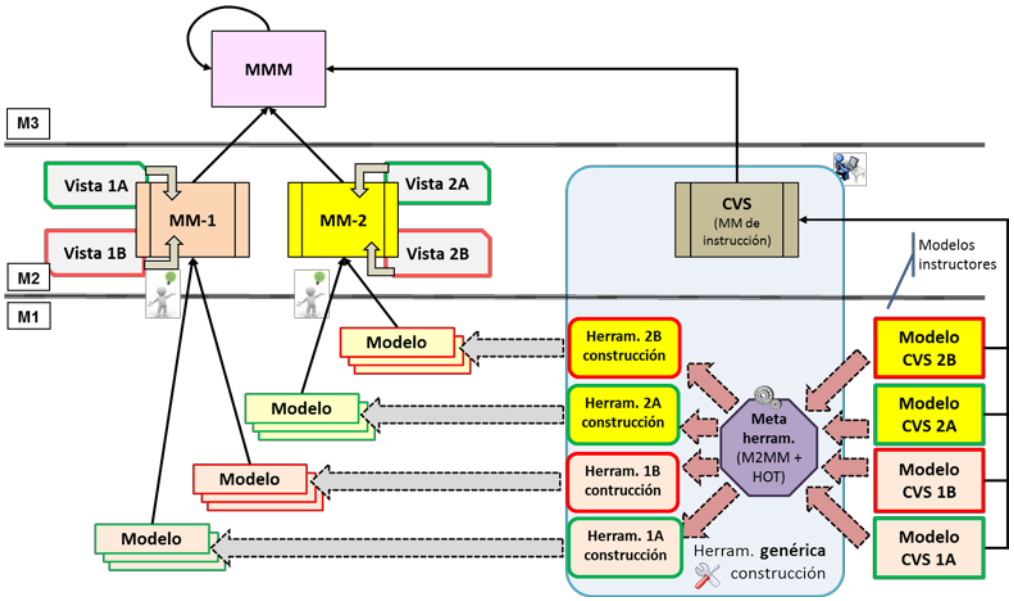


Figura 3.38 – Naturaleza universal de la herramienta diseñada

En la siguiente subsección se expone en detalle la estrategia diseñada. Posteriormente, en la subsección 3.3.2 se presenta el alcance concreto del concepto de vista en el contexto de esta Tesis y una propuesta de metamodelo para soportar la formulación de vistas en forma de modelos.

3.3.1 Construcción de modelos en presencia de vistas

Dada una vista especificada sobre un metamodelo de dominio, se pueden adoptar dos alternativas diametralmente opuestas para crear modelos conformes a él y acordes a ella:

a. **Empleo de herramientas gobernadas por el metamodelo de dominio.** Los modelos acordes a una vista son ante todo modelos conformes al metamodelo de dominio sobre el que se especifica la vista, por lo que pueden ser gestionados utilizando herramientas basadas en él.

   La aplicación de esta estrategia para construcción de modelos se muestra en la Figura 3.39, que ilustra cómo el proceso está gobernado por el metamodelo de dominio y queda bajo responsabilidad del diseñador contemplar y respetar las directrices estipuladas por la vista.

   El modelo construido de esta forma es directamente conforme al metamodelo y acorde a la vista.

[56] Siguiendo la caracterización de vista que se expone en la subsección 3.3.2

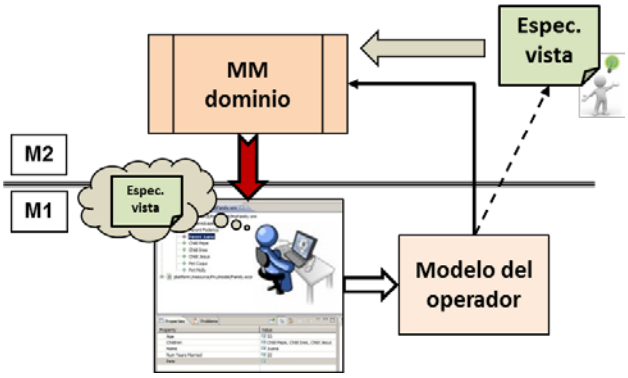


Figura 3.39 – Construcción guiada por el metamodelo de dominio

Para el diseñador, el uso de esta estrategia tiene el inconveniente de que le sigue obligando a trabajar afrontando la complejidad del metamodelo completo, sin ningún tipo de asistencia específica referente a la vista, por lo que la probabilidad de cometer errores es alta. Además, dado que las directrices que definen la vista no están formuladas formalmente, se requiere un trabajo relevante no automatizable para la generación de una herramienta que verifique que los modelos creados son acordes a la vista.

b. **Empleo de herramientas conducidas por la vista**. Se trabaja sobre un metamodelo auxiliar que describe la información específica a la vista, y en base a él, el diseñador construye los modelos acordes a ella. Aunque posiblemente el modelo resultante no sea conforme al metamodelo de dominio y requiera ser transformado a otro que sí lo sea, la ventaja es que el diseñador sólo afronta la complejidad de la vista durante la construcción de modelos.

La estrategia que se propone corresponde a la segunda de las alternativas expuestas y contempla la generación automática del metamodelo relativo a la vista y también de la transformación M2M que produce el modelo final. Por tanto, la estrategia facilita al diseñador la construcción del modelo y deja para una segunda fase transformarlo para que sea conforme al metamodelo de dominio. El metamodelo de soporte se centra en la vista y en la conceptualización que de ella tiene el experto. Por la forma de llegar a él y lo que representa, se le ha denominado metamodelo de *Datos Requeridos por la Vista* (VRD), ya que su objetivo es asistir al diseñador en la creación de modelos acordes a la vista y no formalizar las restricciones que ésta introduce en el metamodelo de dominio.

La Figura 3.40 esquematiza la estrategia propuesta. El diseñador construye modelos conformes al metamodelo VRD correspondiente y a continuación una transformación M2M (*VRD_to_Dominio*) genera el modelo final, conforme al metamodelo de inicio y acorde a la vista.

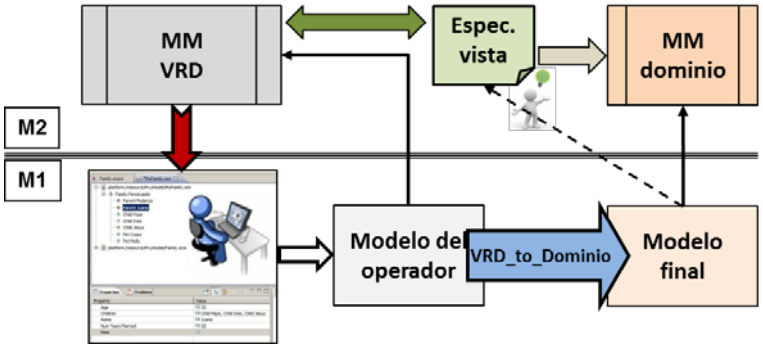


Figura 3.40 – Construcción guiada mediante un metamodelo VRD

De acuerdo con lo expuesto, dado un metamodelo sobre el que se ha especificado una vista, la asistencia al diseñador consiste en proporcionar dos componentes complementarios, no independientes:

- **El metamodelo VRD** en base al que se realiza la introducción de datos.
- **La transformación *VRD_to_Dominio*** que genera el modelo final a partir del modelo introducido por el diseñador.

En lo que resta de esta sección se describe la generación de ambos componentes como salida de sendas transformaciones (una promocionadora y otra HOT) que tienen como entrada la especificación de la vista en forma de modelo junto al propio metamodelo de dominio. Antes de profundizar en dichas transformaciones, lo cual será el objeto de la subsección 3.3.3, la siguiente subsección describe el alcance concreto del concepto *vista* en el contexto de esta Tesis y el metamodelo que da soporte a los modelos de vistas, es decir, el metamodelo de instrucción de la herramienta genérica.

3.3.2 Alcance y formalización de vistas

Una vista es la formalización de un conjunto de condicionantes sobre la información descrita por un metamodelo:

- **Filtrado de clases**. La especificación de las clases del metamodelo cuya instanciación es permitida en los modelos acordes a la vista.
- **Definición de categorías**. La descripción, parcial o completa, de cómo han de ser las instancias de dichas clases permitidas (valores de atributos, referencias nulas, etc.), dando lugar a lo que se ha denominado *categorías* relativas a tales clases permitidas.
- **Objetos obligatorios**. La especificación de las instancias de clases permitidas y acordes a las categorías definidas que han de aparecer obligatoriamente en los modelos.
- **Especificación de ensamblados**. Se define un *ensamblado* como una agrupación de tipos (de entre los permitidos) que se instancian conjuntamente, cada uno de ellos en número concreto y con referencias preestablecidas entre tales instancias.
- **Instancias obligatorias de ensamblados**. La especificación de qué instancias de los ensamblados han de aparecer obligatoriamente en los modelos.

Para dar soporte a la formulación en forma de modelo de las vistas restrictivas, se ha diseñado el metamodelo *Especificación de Vistas Restrictivas* (CVS). En el siguiente apartado se presenta una visión general suya utilizando el lenguaje de metamodelado Ecore.

Metamodelo CVS

El diagrama de clases de la Figura 3.41 muestra el metamodelo CVS, el cual exhibe una estructura convencional, con la clase `CVS_Model` como clase contenedor principal. Ésta define la asociación `referencedMM` mediante la que se referencia al metamodelo sobre el que se especifica la vista. Las otras clases fundamentales son *`Category`* y `ObjectsSpecification`. La primera es una clase abstracta que representa el concepto de *categoría definida por una vista*, bien sobre una clase del correspondiente metamodelo o bien en forma de ensamblado. Por su parte, `ObjectsSpecification` representa el concepto de *elemento (individual o ensamblado) que ha*

de aparecer obligatoriamente en todo modelo acorde a la vista. La clase define los atributos `lowerBound` y `upperBound` que describen el rango de cardinalidad de tales elementos.

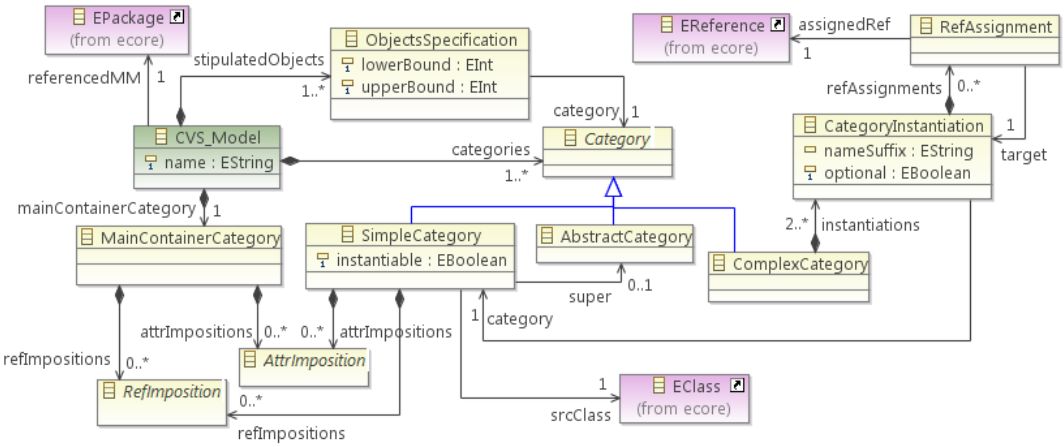


Figura 3.41 – Metamodelo CVS

La clase `CVS_Model` define dos composiciones: `categories` y `stipulatedObjects`. A través de la primera, el contenedor principal de un modelo CVS contiene la descripción de aquellas categorías definidas por la vista, mientras que por medio de la segunda, la descripción de los elementos obligatorios. Por su parte, `ObjectsSpecification` define la asociación `category` mediante la que tales objetos estipulados indican la categoría, de entre las especificadas por la vista, respecto a la que han de ser acordes.

Categorías. El metamodelo CVS define tres subclases de *`Category`*.

- **`SimpleCategory`**. Representa el concepto de *categoría básica definida por una vista sobre una clase (permitida) del correspondiente metamodelo*. Esta clase define la asociación `srcClass` que permite a sus instancias referenciar a la correspondiente clase base, así como las composiciones `attrImpositions` y `refImpositions` por medio de las cuales una categoría simple contiene las descripciones de aquellas imposiciones que la vista define sobre las propiedades de la clase base. Además, mediante el atributo `instantiable` un objeto suyo declara si representa a una categoría instanciable, esto es, una categoría tal que en un modelo acorde a la vista pueden existir elementos individuales acordes a ella. En caso de que no, esto significa que únicamente podrán aparecer como parte de instancias de ensamblados.
- **`AbstractCategory`**. Clase que se introduce con el objetivo de contemplar el caso de que existan categorías simples definidas sobre clases que compartan superclase.
- **`ComplexCategory`**. Representa el concepto de *ensamblado definido por una vista*. Esta clase define la composición `instantiations` para especificar los elementos integrantes del ensamblado.

En la formulación de la constitución de un ensamblado participan las clases `CategoryInstantiation` y `RefAssignment`.

- **`CategoryInstantiation`**. Representa el concepto de instanciación particular de una determinada categoría simple en un ensamblado. La clase define una asociación `category` mediante la que sus objetos indican la categoría simple en cuestión. También define los

atributos `optional` y `nameSuffix`, para especificar respectivamente si tal instanciación es opcional dentro del ensamblado y para declarar un posible sufijo literal. Por último, la clase define la composición `refAssignments` por medio de la cual sus instancias pueden albergar objetos `RefAssignment`. Gracias a ello se cubre el hecho de que en un ensamblado puedan estar preestablecidos enlaces entre los propios integrantes o incluso entre elementos de diferentes ensamblados.

- **`RefAssignment`**. Representa un mecanismo para el establecimiento de una referencia de un elemento integrante de un ensamblado hacia otro elemento del mismo ensamblado o de otro distinto. La clase define dos asociaciones: `assignedRef` y `target`. Mediante la primera, sus instancias apuntan a la referencia que se desea establecer y mediante la segunda al elemento destino sobre el que queda establecido el enlace.

En un modelo CVS no se han de especificar instancias de `SimpleCategory` sobre la clase contenedor principal del metamodelo de dominio. En su lugar, el metamodelo CVS presenta una clase más, la clase `MainContainerCategory`, que representa la descripción parcial o completa de cómo ha de estar configurado el contenedor principal en un modelo acorde a la vista. Las dos composiciones que define son análogas a las del mismo nombre en la clase `SimpleCategory`. La clase `CVS_Model` define una composición más, `mainContainerCategory`, a través de la cual el contenedor principal de un modelo CVS contiene la única instancia de esta clase `MainContainerCategory`, cuya definición explícita por separado es una decisión de diseño con el propósito de facilitar el desarrollo de la metaherramienta expuesta en la subsección siguiente.

Imposiciones sobre atributos y sobre referencias. Las imposiciones que una vista establece sobre las propiedades de una clase (permitida) al definir una categoría simple sobre ella vienen representadas mediante instancias de *`AttrImposition`* y *`RefImposition`*. Se contemplan dos tipos de imposiciones sobre un atributo (asignación de un valor fijo e imposición de que tenga igual valor que otro atributo) y tres tipos de imposiciones sobre una referencia (especificación de una categoría respecto a la que ha de ser acorde su *target*, la obligación de ser *null* o la imposición de que tenga igual *target* que otra referencia). Esta variedad se representa respectivamente por las clases `ValueAssignment` y `EqualizationWithOtherAttr` (subclases de *`AttrImposition`*) y `TypeSpecification`, `Nullification` y `EqualizationWithOtherRef` (subclases de *`RefImposition`*). La Figura 3.42 muestra las clases mencionadas

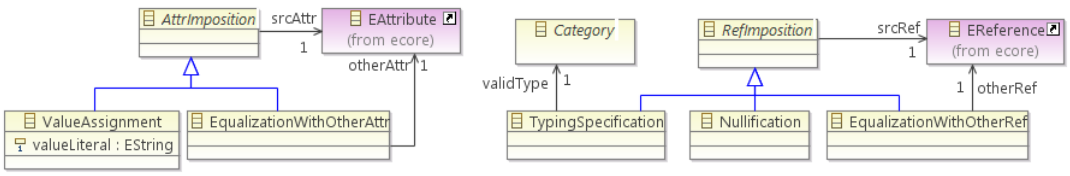


Figura 3.42 – Clases relativas a imposiciones sobre atributos y sobre referencias

3.3.3 Herramienta genérica para construcción de modelos

En la subsección 3.3.1 se ha expuesto el diseño de una estrategia para facilitar la construcción de modelos acordes a una vista, estrategia que requiere desarrollar dos componentes (metamodelo VRD y transformación *VRD_to_Dominio*) propios de la vista en cuestión (y por extensión del metamodelo de dominio).

Se aborda en esta subsección el diseño de la herramienta genérica, aplicable a cualquier metamodelo de dominio y a cualquier vista especificada sobre él. Su carácter genérico se consigue en forma de metaherramienta que genera bajo demanda la herramienta específica correspondiente.

Diseño de la metaherramienta

La Figura 3.43 muestra cómo la metaherramienta opera en dos pasos. A partir de la formulación de la vista conforme al metamodelo CVS, genera sucesivamente los dos componentes de cada herramienta específica.

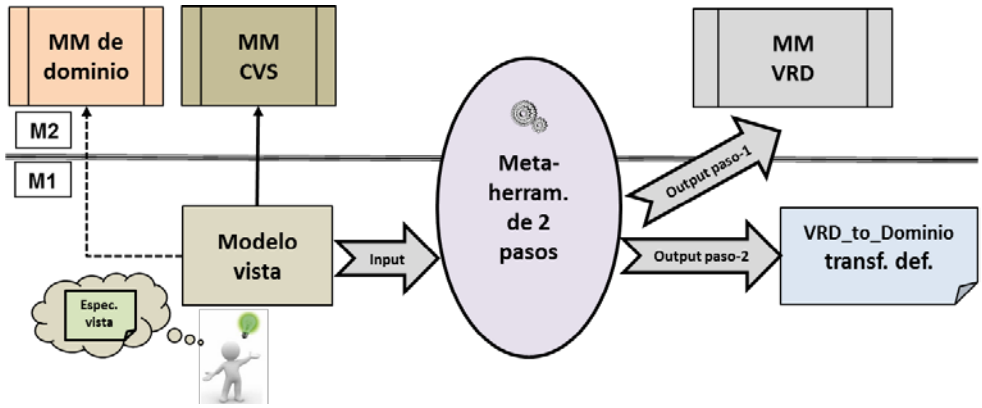


Figura 3.43 – Metaherramienta de dos pasos

1. **Generación del metamodelo VRD** que dirige la construcción restringida de modelos.
   Tal y como se muestra en la Figura 3.44, este componente se obtiene a partir del modelo CVS de la vista a través de la transformación promocionadora *CVS_to_Ecore*.

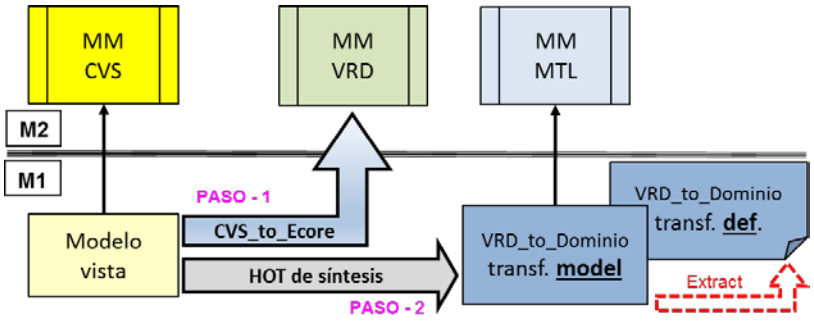


Figura 3.44 – Generación del metamodelo VRD y de la transformación VRD_to_Dominio

2. **Generación de la transformación *VRD_to_Dominio*** que convierte los modelos de datos requeridos a modelos conformes al metamodelo de dominio.
   Según se muestra en la Figura 3.44, a partir del modelo CVS de la vista se hace uso de la técnica HOT para obtener la transformación *VRD_to_Dominio* en forma de modelo conforme al metamodelo de ATL.

   En este caso se trata de una HOT de tipo síntesis y el modelo de transformación generado es posteriormente extraído a la notación textual de ATL. Evidentemente, la transformación obtenida ha de ser adecuada a la estructura específica del metamodelo VRD generado en el primer paso.

Transformaciones involucradas

En resumen, la estrategia diseñada involucra diversas transformaciones de modelos. En primer lugar, dado una vista especificada sobre un metamodelo de dominio, los modelos reducidos construidos por el diseñador son transformados a los modelos definitivos mediante la correspondiente transformación *VRD_to_Dominio*. En lo relativo a la generación automática de los componentes correspondientes a cada situación (metamodelo VRD y la propia transformación *VRD_to_Dominio*), se utilizan respectivamente la transformación promocionadora *CVS_to_Ecore* y la HOT *CVS_to_MTL*, donde MTL es el lenguaje de transformación empleado (ATL en este caso).

3.3.4 Vistas restrictivas en MAST-2

El entorno MAST-2 representa un banco de trabajo ideal sobre el que implementar la estrategia diseñada para facilitar la construcción de modelos acordes a vistas restrictivas, puesto que, con el propósito de poder modelar el comportamiento temporal de un amplio espectro de SDTREs, el metamodelo MAST-2 está constituido por un total de 143 clases. Aunque su complejidad está justificada, ya que ha de cubrir los modelos sobre los que se aplican las más de 10 herramientas del entorno MAST (análisis de planificabilidad, asignación de prioridades, cálculo de holguras, etc.), representa un hándicap para el diseñador que usa una metodología y tecnología concretas y que sólo necesita una fracción muy reducida de MAST-2 para desarrollar sus sistemas.

La vista LinuxClassicRMA (LCRMA)

Como ejemplo, se considera el caso de una empresa que desarrolla software de tiempo real para un equipo que integra una determinada plataforma RT-Linux. Además, por la naturaleza del software requerido, se utiliza la técnica de análisis RMA clásica [136] para analizar su planificabilidad. En este caso, el metamodelo de dominio es MAST-2 y sobre él se define la vista *LinuxClassicRMA* (LCRMA), que lo delimita al caso del software que desarrolla la empresa citada, consecuencia de la plataforma de ejecución empleada y de la herramienta de análisis utilizada:

- **Restricciones por la plataforma de ejecución utilizada**: El que el procesador utilizado sea una determinada plataforma RT-Linux delimita drásticamente el modelo de la plataforma de ejecución del sistema, el cual contiene:
  - Un único elemento procesador, cuyos atributos toman un valor fijo conocido.
  - Un único planificador, asociado al procesador de la plataforma. Su política de planificación es de prioridades fijas (FP) y el rango de prioridades que es posible asignar es de [0,100]. En consecuencia, todos los *threads* que se definan en un modelo tienen parámetros de planificación FP y son planificados por este único planificador de la plataforma.
  - Un único reloj, que es el asociado al único procesador de la plataforma.
  - Un único tipo de elemento de sincronización entre los *threads*, que son los mutexes con protocolo de techo inmediato de prioridad.
- **Restricciones por la naturaleza del software desarrollado**: La reactividad de las aplicaciones se compone de un conjunto de tareas, donde cada una:
  - Tiene una activación periódica temporizada por el reloj del sistema.
  - Se ejecuta en un *thread* propio.

- Puede tomar recursos compartidos al inicio de su ejecución, que han de ser liberados al finalizar la misma.
- Puede tener un requisito de plazo temporal estricto asociado a la finalización de su ejecución y relativo a la activación.

El objetivo de la vista LCRMA es liberar al diseñador de aplicaciones que quiere analizar la planificabilidad y asignar prioridades a las aplicaciones de un nuevo software, no sólo de tener que conocer y decidir sobre las 143 clases de MAST-2, sino incluso de tener que conocer las 17 clases que se necesitan para modelar el software con las restricciones de la empresa.

Un modelo MAST-2 acorde a LCRMA

De acuerdo con los puntos reseñados, la Figura 3.45 muestra una ilustración, a modo de diagrama de objetos, con un ejemplo de modelo acorde a la vista LCRMA. La Figura 3.46 muestra una visión reactiva de más alto nivel.

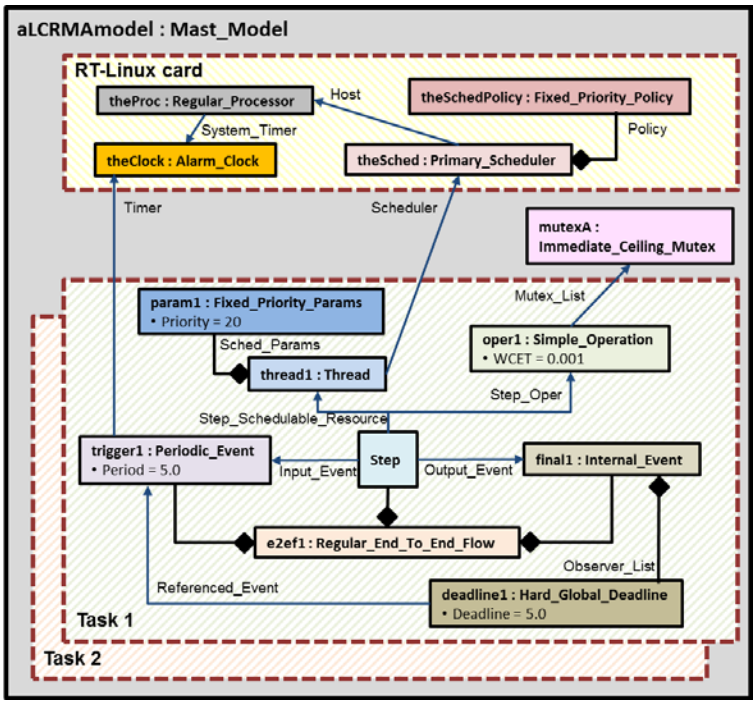


Figura 3.45 – Ejemplo de modelo LCRMA

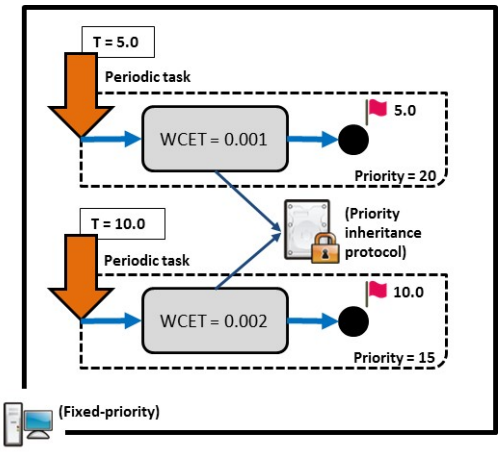


Figura 3.46 – Visión reactiva del ejemplo de modelo LCRMA

La vista LCRMA formulada como modelo CVS

La Figura 3.47 muestra una parte del modelo CVS representativo de la vista LinuxClassicRMA.

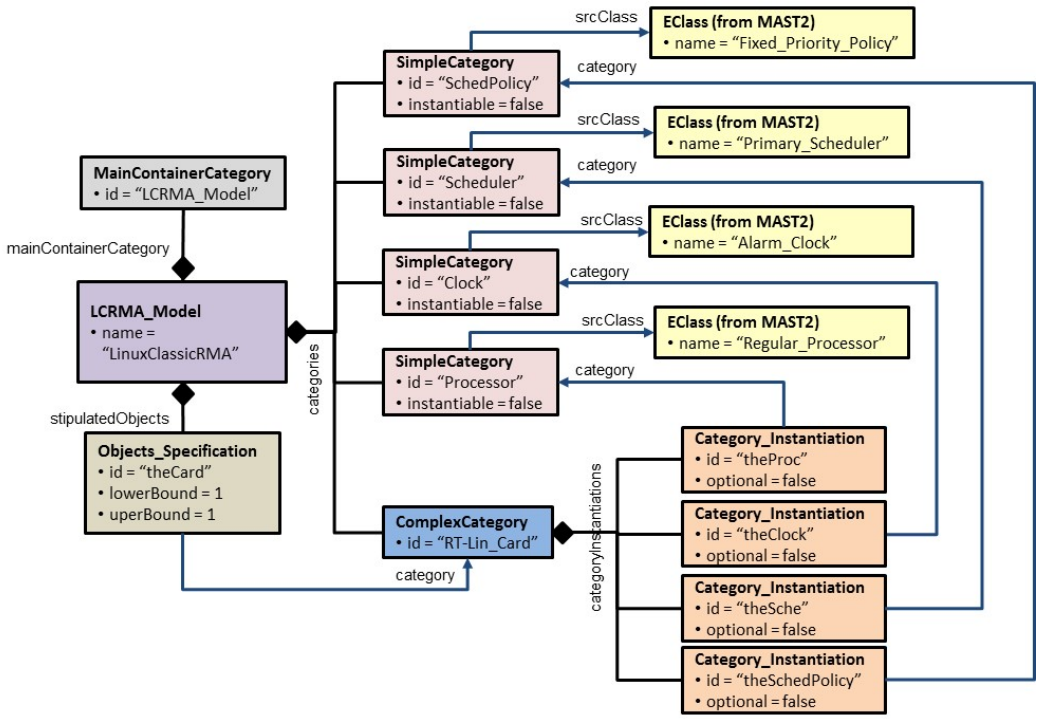


Figura 3.47 – Parte del modelo CVS representativo de la vista LCRMA

La ilustración se centra en cómo se modela el hecho de que en todo modelo LinuxClassicRMA sólo ha de haber una plataforma RT-Linux, basada en una política de planificación FP. El ensamblado *RT-Lin_Card* se define como una categoría compleja que aglutina una instanciación de cada una de las siguientes categorías simples: `Processor`, `Clock`, `Scheduler` y `SchedPolicy`. Éstas se definen respectivamente sobre las clases `Regular_Processor`, `Alarm_Clock`, `Primary_Scheduler` y `Fixed_Priority_Policy` del metamodelo MAST-2. Finalmente, la tarjeta se declara como una instancia `ObjectsSpecification`, que apunta a la categoría compleja definida y que impone *theCard* como nombre y que la instancia es única.

La Figura 3.48 muestra en detalle la configuración de las categorías `SchedPolicy` y `Scheduler`. La primera ilustra cómo se imponen valores fijos a atributos mientras que la segunda ilustra cómo se establecen enlaces entre los integrantes del ensamblado.

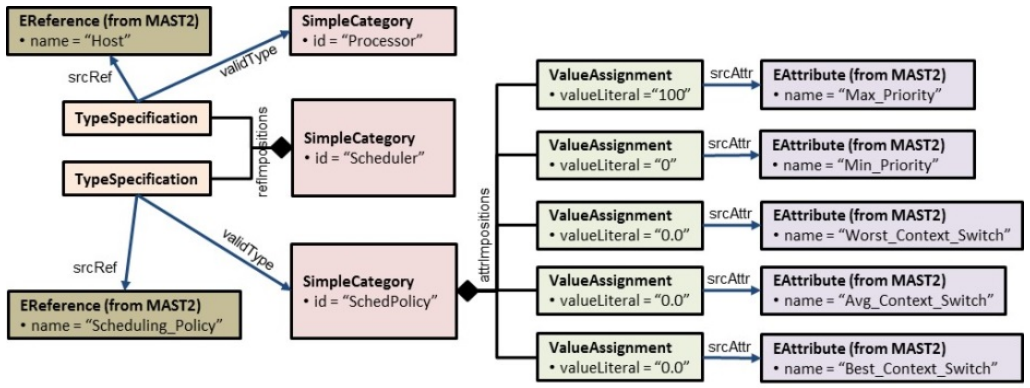


Figura 3.48 – Detalle de las categorías simples `Scheduler` y `SchedPolicy`

Metamodelo VRD_for_Mast2-LCRMA

El diagrama de clases mostrado en la Figura 3.49 representa el metamodelo VRD correspondiente a la vista LCRMA. El diseñador que afronta la tarea de crear modelos LCRMA únicamente debe construir modelos conformes a este metamodelo reducido, los cuales posteriormente serán transformados a modelos MAST-2 (y acordes a LCRMA) mediante la transformación *VRD_for_Mast2-LCRMA_to_MAST2* (particularización de *VRD_to_Dominio* de la Figura 3.43).

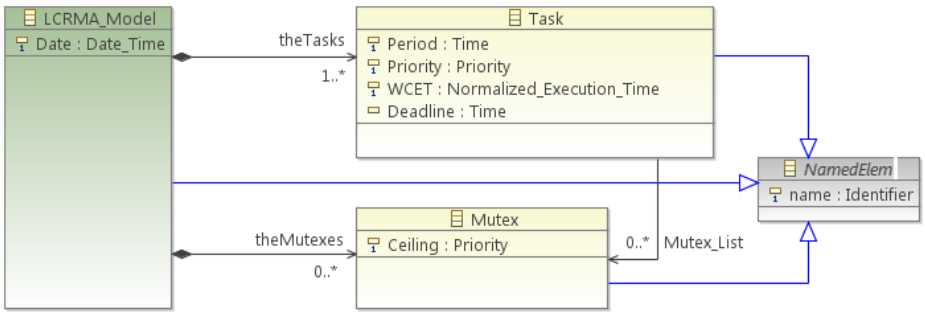


Figura 3.49 – Metamodelo VRD_for_Mast2-LCRMA

Modelo del operador

La Figura 3.50 muestra el modelo (conforme a VRD_for_Mast2-LCRMA) que ha de construir el diseñador para obtener el modelo MAST-2 de la Figura 3.45. La diferencia de tamaño y complejidad conceptual es sustancial.

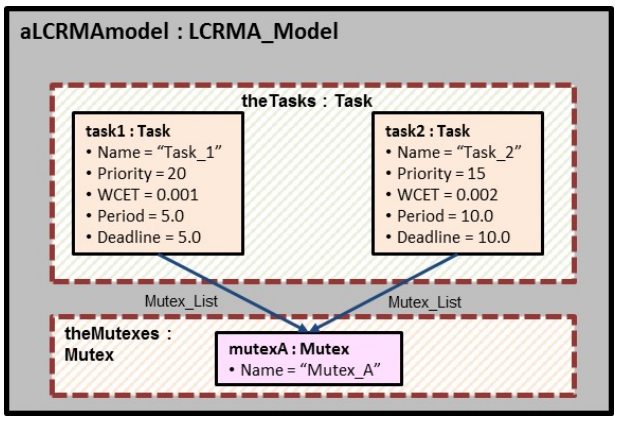


Figura 3.50 – Modelo construido por el diseñador

3.3.5 Enfoques relacionados

El concepto de vista posee una larga tradición en el área de las BBDD [137]. En la literatura MDSE pueden encontrarse numerosas contribuciones proponiendo desarrollos basados en adoptar y adaptar este concepto al área del Modelado. En primer lugar, trabajos que exhiben como motivación el problema de procesos MDD que usan varios metamodelos y, consecuentemente, tienen que manejar información diseminada en modelos heterogéneos interrelacionados. Propuestas recientes en esta dirección son las metodologías *EMF-Views* [138] y *Flexible Views* [139], las cuales, con el propósito de reducir la complejidad, proponen gestionar los modelos por medio de vistas parciales especializadas que seleccionan y/o agregan la información de tales modelos heterogéneos. La principal diferencia con la estrategia aquí propuesta es que está

orientada a la construcción de modelos según una estrategia dirigida por la propia vista, mientras que esos enfoques están orientados a la creación de vistas a ser aplicadas para visualización en base a filtrar y/o combinar elementos de modelos heterogéneos ya existentes. Más cercanas son las propuestas de [140] y [141], aunque en ellas las vistas son meras porciones del metamodelo de dominio, con lo que la estrategia para construcción de modelos no está específicamente definida por la especificación de la vista.

3.4 Herramienta para Interoperabilidad XML ↔ Modelware

De acuerdo con el mapa conceptual planteado en la sección 2.1, la tecnología de soporte de los entornos MDSE proporciona un *framework* estándar de interoperabilidad entre los diferentes espacios tecnológicos que soporta. Si se utiliza este *framework,* la tecnología del entorno proporciona las herramientas para establecer las correspondencias entre las formulación de un dominio en los diferentes espacios. Así, la tecnología EMF/Eclipse proporciona recursos para realizar conversiones automáticas entre las representaciones de un dominio por su metamodelo, por su DSL proporcionado por Xtext y su representación XMI.

Sin embargo, este escenario no siempre es el que se presenta. Por la costumbre o tradición del dominio de aplicación o por la necesidad de colaborar con otros MDSE puede ocurrir que la representación textual de dominio o la secuenciación XML que se usan, sean legados y diferentes de los considerados como estándares en el *framework.* En este caso, la operatividad del entorno requiere desarrollar herramientas de interoperabilidad específicas para ese dominio, lo cual exige mucho trabajo y complica el mantenimiento frente a cambios del dominio. En esta sección, se propone el uso de herramientas genéricas como forma de facilitar el desarrollo y mantenimiento de las herramientas de interoperabilidad no estándar entre espacios tecnológicos.

Escenario de interoperabilidad XML ↔ Modelware que precisa elaboración

El caso que se aborda es la interoperabilidad en un cierto dominio entre una formalización particular no estándar del espacio tecnológico XML y el espacio tecnológico Modelware estándar con Ecore. Esto se presenta si ambas formalizaciones (W3C-Schema y metamodelo) han sido llevadas a cabo de forma independiente, por ejemplo el W3C-Schema con anterioridad al metamodelo. De esta manera, entre ellas cabe hablar de una posible desalineación[57], que puede ir desde una cuestión de mera nomenclatura hasta aspectos estructurales profundos. La Figura 3.51 muestra este escenario.

La flecha bidireccional punteada indica equivalencia conceptual entre el W3C-Schema y el metamodelo, pero habiendo sido ambos desarrollados independientemente y por tanto, existiendo una más que probable desalineación.

Puesto que la interoperabilidad XML ↔ Modelware a nivel M1 proporcionada por defecto por EMF es válida para cualquier dominio cuyas formalizaciones, metamodelo y W3C-Schema, se encuentren ligadas en términos EMF, esto es, obtenidas una a partir de la otra, en este escenario no puede hacerse uso de tal interoperabilidad. Sin embargo, otras vías para construir pasarelas a nivel M1 son posibles.

[57] Pensar, por ejemplo, en que una pareja de metamodelo y W3C-Schema obtenidos uno a partir del otro en EMF están lo que podría llamarse “completamente alineados”.

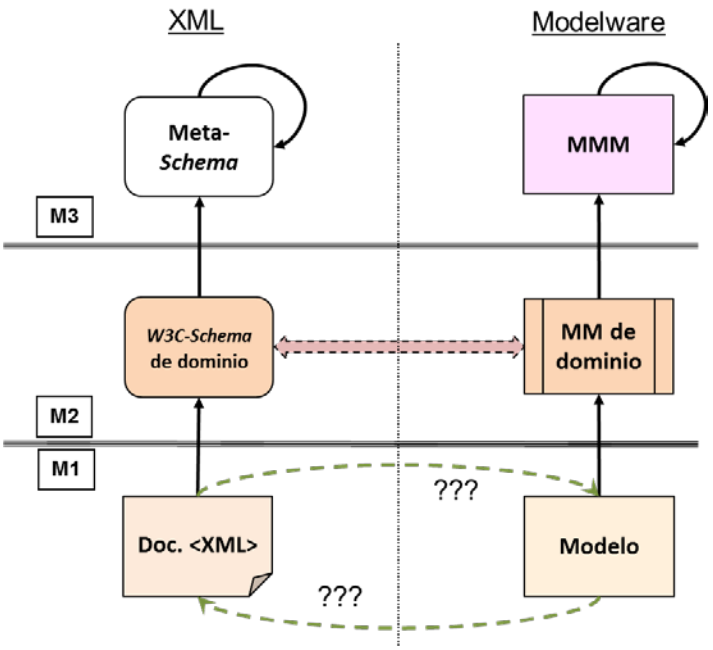


Figura 3.51 – Formalizaciones de un dominio en XML y Modelware acometidas de forma independiente

Motivación y conveniencia de la herramienta genérica contribuida

Esta sección contribuye al desarrollo de herramientas para interoperabilidad entre espacios tecnológicos. En concreto, se aborda el diseño de una herramienta genérica para conseguir interoperabilidad en el escenario expuesto en el apartado anterior. Así, la Figura 3.52 toma como base la Figura 2.4 para reseñar la ubicación de la interoperabilidad abordada dentro de un puente MDI (flechas rojas). Se da soporte a las etapas extremas del puente, esto es, las proyecciones hacia y desde Modelware.

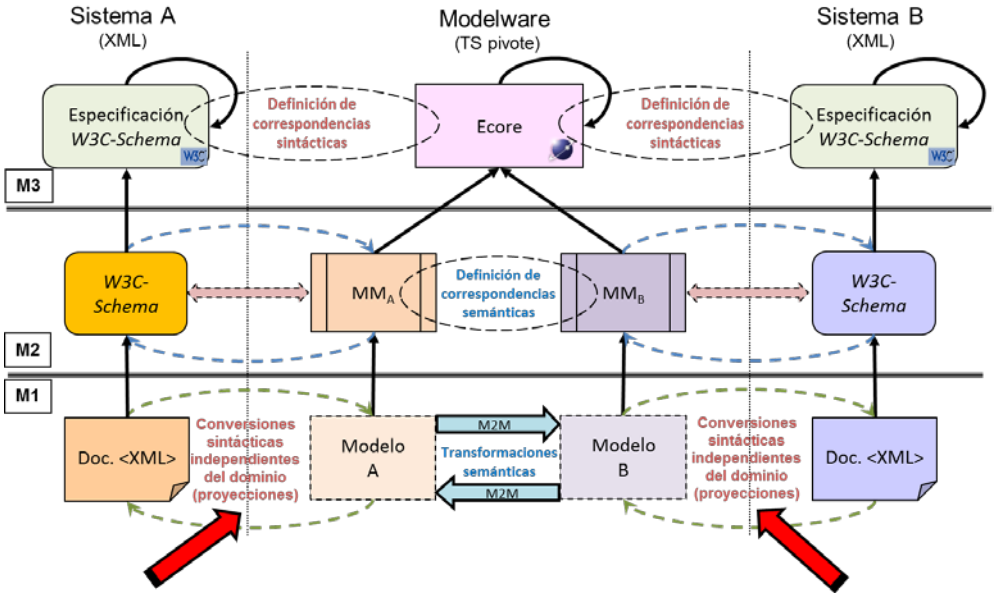


Figura 3.52 – Ubicación de la interoperabilidad abordada

La herramienta implementa una estrategia *model-driven.*

- **Proyección XML → Modelware**. En este sentido la estrategia está compuesta por una sucesión de:

i. **Conversión de documento XML a un modelo** intermedio, conforme a un metamodelo propio de la herramienta.

ii. **Transformación M2M** para obtener el modelo conforme al metamodelo de dominio.

Tomando como TS base a Modelware, cabe calificar a esta proyección como *inyección XML*. Es más, aunque la proyección englobe ambos pasos, puesto que es realmente en el primero de ellos donde se produce el cruce entre espacios (el segundo se realiza ya en el seno de Modelware), se reserva el término *inyección XML* a la conversión de documento XML a modelo intermedio.

- **Proyección Modelware → XML**. En este sentido la estrategia está compuesta por una sucesión de:

  i. **Transformación M2M** para obtener modelo intermedio.

  ii. **Conversión de modelo a documento XML**.

  Consideraciones análogas respecto a la expresión *extracción XML*.

Si la estrategia se implementase mediante herramientas MDSE estándar, sería necesario el desarrollo *ad hoc* de una herramienta específica para cada caso, esto es, para cada dominio formalizado mediante una pareja de metamodelo y W3C-Schema desarrollados independientemente. Ésta sería una solución no deseable, pues:

- **La evolución del dominio** implicaría la modificación de ambos artefactos, con la consiguiente necesidad de rediseñar la correspondiente herramienta de interoperabilidad.
- **La incorporación a un entorno de un nuevo dominio** requeriría la creación de una nueva herramientas de interoperabilidad.

Por todo ello, se propone una herramienta genérica basada en una metaherramienta que bajo demanda genere cada herramienta específica de interoperabilidad e invoque su ejecución en el sentido de la proyección necesaria en cada caso. Puesto que éstas van a adoptar la forma de una pareja de transformaciones M2M (la inyección y la extracción XML vienen proporcionadas por terceras partes), la metaherramienta está basada en sendas HOTs.

Herramienta basada en metaherramienta

La Figura 3.53 muestra los elementos básicos de la herramienta diseñada. La caracterización y diseño como herramienta genérica requiere la definición del ECC que describe su funcionalidad y su consiguiente formalización como metamodelo de instrucción. El ECC se tratará en la subsección 3.4.2, aunque para facilitar la comprensión de la Figura 3.53 se indica que el metamodelo *Mapping* constituye el metamodelo de instrucción y cada modelo instancia suya describe las correspondencias entre metamodelo y W3C-Schema.

Dada una pareja de metamodelo y W3C-Schema, la formulación de un modelo descriptivo de las correspondencias entre sus elementos instruye a la metaherramienta (basada en dos HOTs), núcleo de la herramienta genérica de interoperabilidad, para que produzca la herramienta específica adecuada (basada en dos transformaciones M2M).

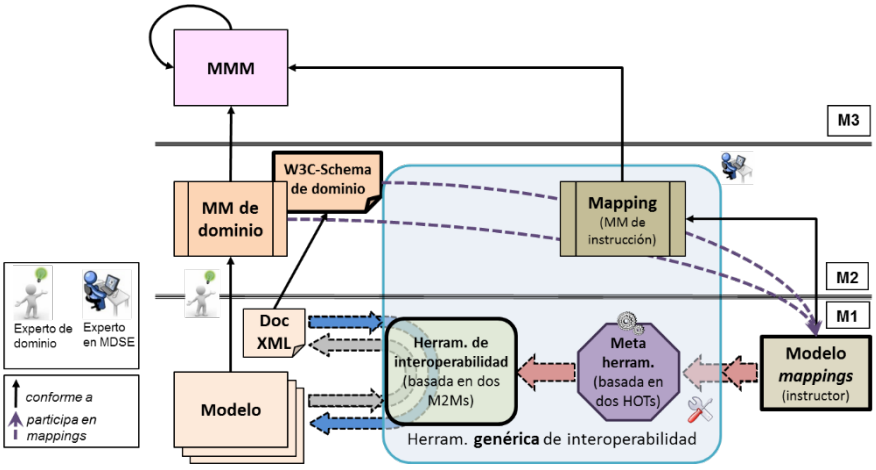


Figura 3.53 – Esquema de la herramienta genérica de interoperabilidad

Así, la actuación ante cualquier supuesto dentro de la casuística evolutiva enumerada anteriormente se reduce a la formulación del modelo instructor que caracteriza a la nueva (renovada o inédita) relación entre ambos artefactos. A partir de este modelo se genera la nueva herramienta específica de interoperabilidad, sin necesidad de adaptación de la que ha quedado obsoleta, la cual puede desecharse.

Las siguientes subsecciones exponen en detalle la solución de interoperabilidad diseñada y su automatización.

3.4.1 Solución de interoperabilidad

Tomando como *framework* base EMF, AMMA facilita una vía alternativa para interoperabilidad entre ambos espacios, mucho más flexible que la proporcionada por defecto por EMF. Esto posibilita desarrollar puentes de interoperabilidad a nivel M1 según la estrategia que se expone en los siguientes apartados.

Infraestructura proporcionada por AMMA

AMMA proporciona una infraestructura consistente en [94, 95]:

a. **El metamodelo XML.ecore**. Se trata de un metamodelo muy sencillo pero que comprende todo lo necesario para modelar el contenido de cualquier documento XML. La Figura 3.54 muestra la estructura del metamodelo.

b. **Proyectores**. Los ATP (*ATLAS Technical Projector*) son mecanismos capaces de comunicar los espacios XML y Modelware convirtiendo cualquier documento XML a modelo XML (conforme al metamodelo XML) y simétricamente, convirtiendo cualquier modelo XML a su serialización en forma de documento XML.

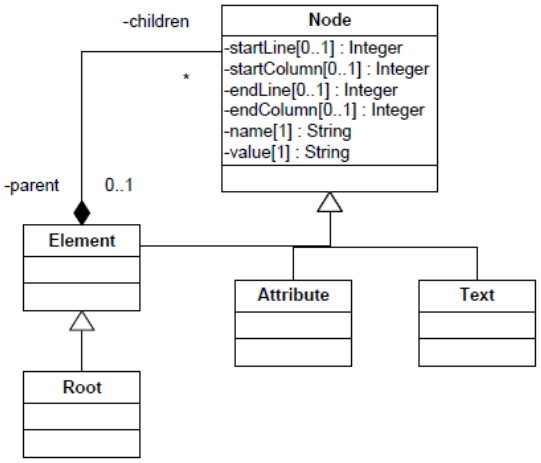


Figura 3.54 – Metamodelo `XML.ecore`

La Figura 3.55 ilustra la interoperabilidad XML ↔ Modelware gracias a esta infraestructura.

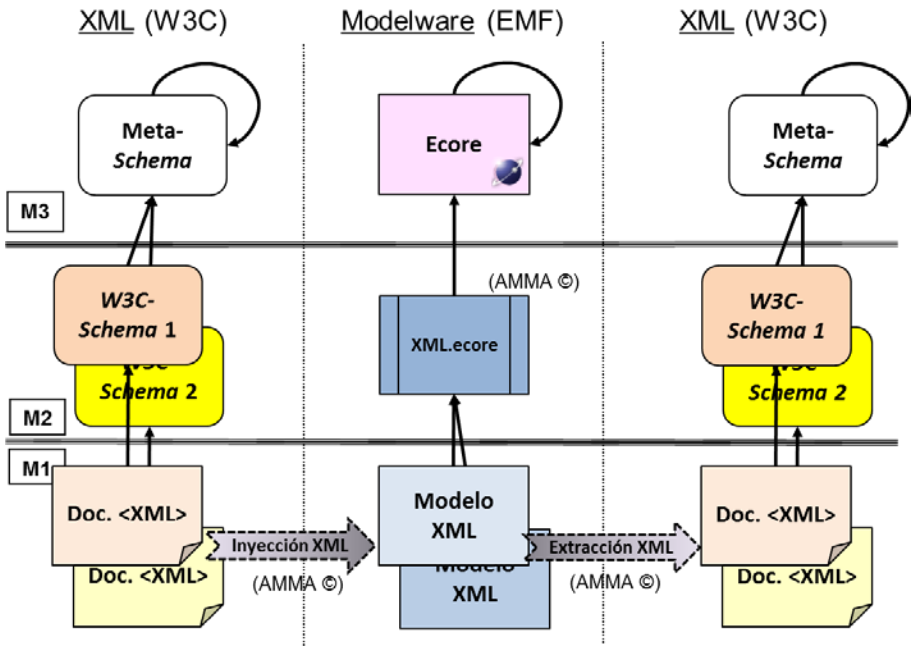


Figura 3.55 – Base operativa facilitada por AMMA para interoperabilidad XML ↔ Modelware

Con estos recursos, la idea central de la estrategia de interoperabilidad desarrollada consiste en superar la desalineación entre W3C-Schema y metamodelo en el seno de Modelware, mediante sendas transformaciones M2M puras. Éstas generan el modelo conforme al metamodelo de dominio a partir del modelo XML inyectado y viceversa. La responsabilidad de cruzar espacios se delega por tanto a los ATPs. Los siguientes apartados detallan el proceso en ambos sentidos.

Conversión de documento XML a modelo

La Figura 3.56 muestra el escenario en el que se obtiene la representación de un sistema en forma de modelo conforme al metamodelo de dominio a partir de su formulación como documento XML.

Como puede observarse, el proceso T2M se desglosa en dos pasos utilizando como pivote la representación de la información del documento XML formulada de acuerdo con el metamodelo `XML.ecore` anteriormente presentado:

1. **Transformación T2M** (*inyección XML*) por la que se obtiene un modelo conforme al metamodelo `XML.ecore`.
2. **Transformación M2M** (*XML_to_MM*) que genera el modelo final.

En la implementación de la transformación M2M es donde el desarrollador ha de tener plenamente en consideración las correspondencias entre los elementos del metamodelo de dominio y las construcciones en el W3C-Schema de partida. De hecho, la implementación es completamente dependiente de las estrategias de formalización empleadas en el W3C-Schema.

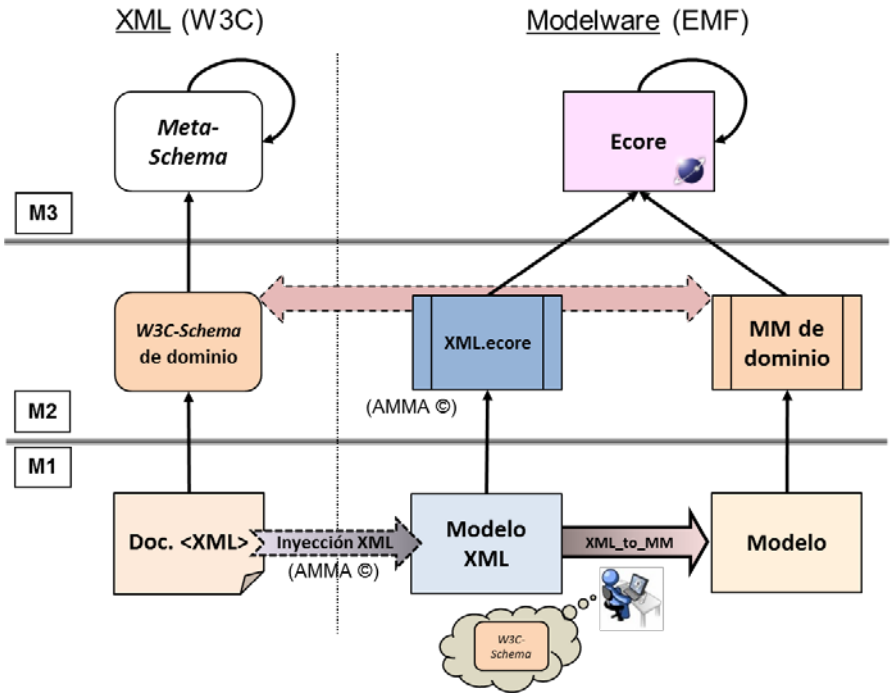


Figura 3.56 – Proceso de conversión de documento XML a modelo

Conversión de modelo a documento XML

La Figura 3.57 muestra el escenario en el que se obtiene la formulación de un sistema como documento XML a partir de su representación en forma de modelo conforme al metamodelo de dominio.

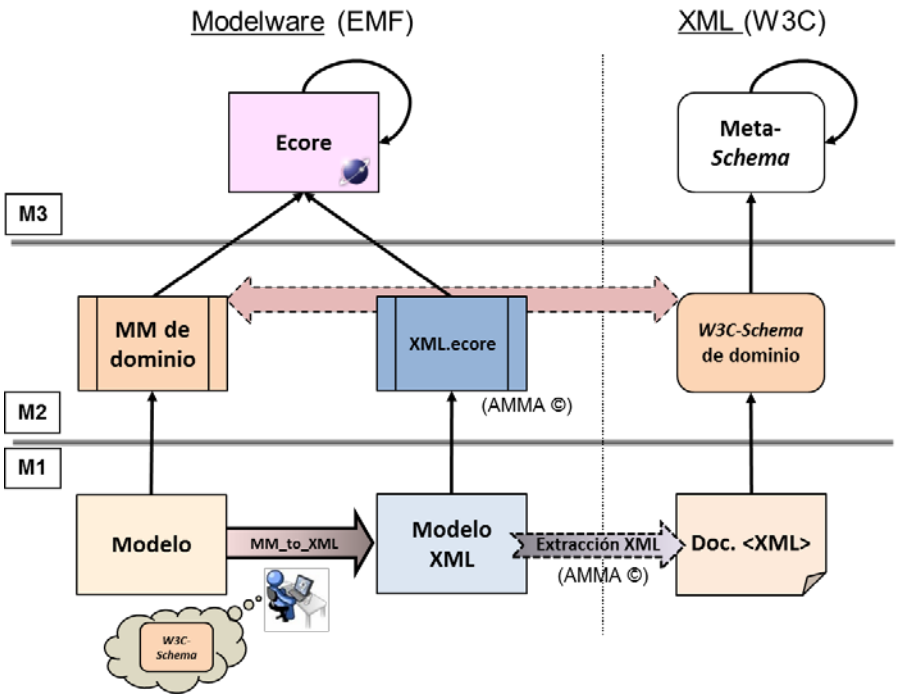


Figura 3.57 – Proceso de conversión de modelo a documento XML

De forma análoga al caso anterior, lo que inicialmente cabe contemplar como un proceso M2T, se desglosa en una cadena compuesta de dos pasos:

1. **Transformación M2M** (*MM_to_XML*) por la que se obtiene un modelo conforme al metamodelo `XML.ecore`.
2. **Transformación M2T** (*extracción XML*) por la que se obtiene el documento XML final.

De nuevo, en la implementación de la transformación M2M es donde el desarrollador ha de tener plenamente en consideración las correspondencias entre los elementos del metamodelo de dominio y las construcciones en el W3C-Schema. Al igual que en el caso anterior, la implementación es completamente dependiente de las estrategias de formalización empleadas en el W3C-Schema de dominio.

Tal y como muestra la Figura 3.58, el análisis de la forma en que las diversas construcciones de metamodelado presentes en el metamodelo de dominio se corresponden con las distintas técnicas de formalización empleadas en el W3C-Schema permite la implementación de las transformaciones, dependiente en ambos casos de tales correspondencias.

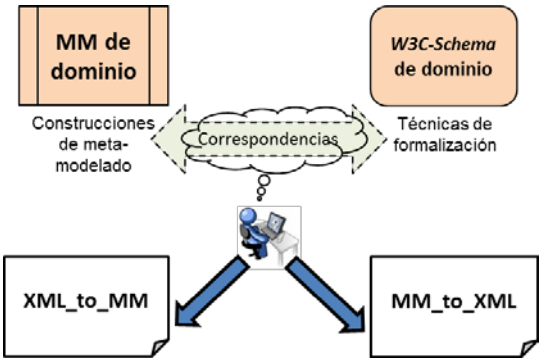


Figura 3.58 – Implementación de las transformaciones M2M de inyección y extracción

3.4.2 Automatización de la solución diseñada

En la estrategia de interoperabilidad expuesta, para cada par de W3C-Schema y metamodelo, las correspondencias entre elementos de ambos artefactos quedan plasmadas en la codificación de las reglas que componen las transformaciones *XML_to_MM* y *MM_to_XML*. Tal y como muestra la Figura 3.59, la formulación de tales correspondencias o *mappings* en forma de modelo (modelo instructor) hace posible la generación automática de ambas transformaciones, es decir, de la herramienta de interoperabilidad para cada caso concreto.

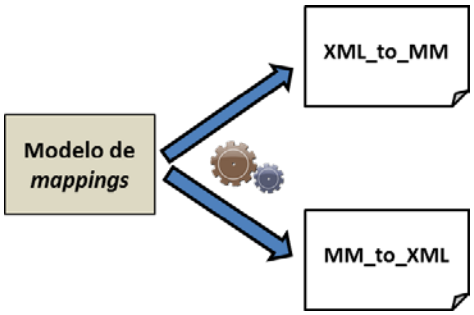


Figura 3.59 – Generación automática de las transformaciones M2M

Como muestra la Figura 3.60, tal modelo de *mappings* ha de tener visibilidad sobre los artefactos a relacionar, esto es, el metamodelo y el W3C-Schema de dominio. En lo relativo a este último, para permitir su importación es necesaria su formulación equivalente en forma de modelo.

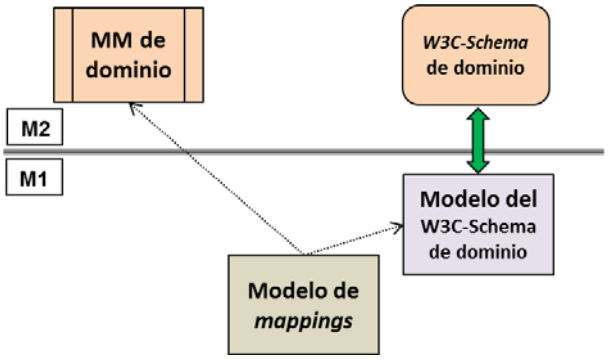


Figura 3.60 – Modelo de *mappings*

La Figura 3.61 muestra las dos HOT de síntesis que constituyen el núcleo de la metaherramienta. Ambas toman como entrada el modelo de *mappings* y generan los modelos de transformación requeridos, los cuales son finalmente serializados a la sintaxis textual del lenguaje de transformación empleado (ATL).

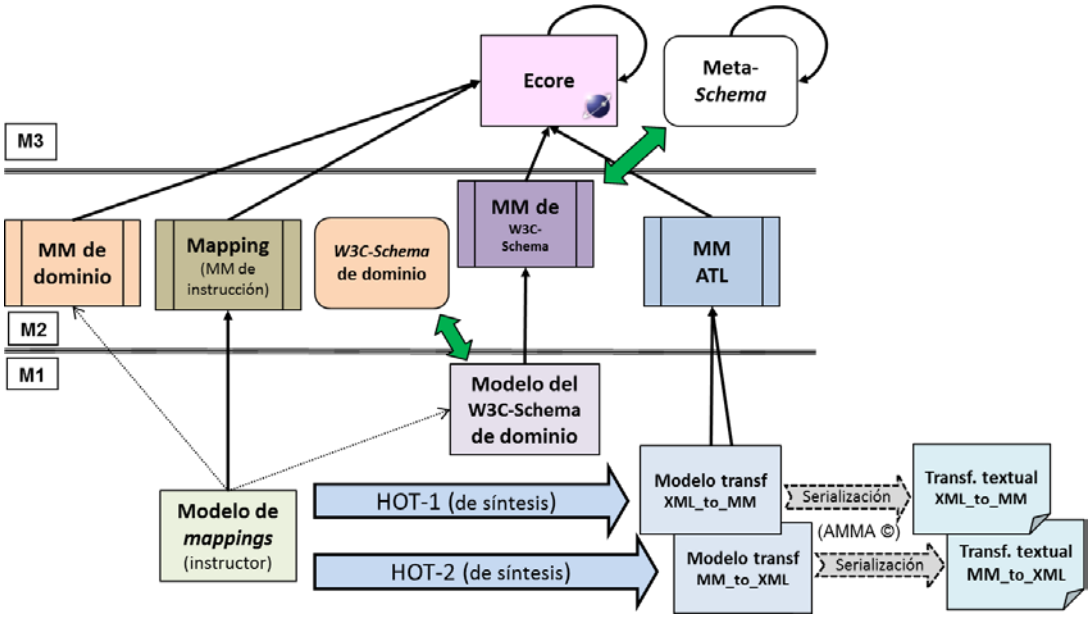


Figura 3.61 – Esquema de la metaherramienta de interoperabilidad XML ↔ Modelware

Además de ser necesario el empleo de un lenguaje de transformación de modelos que permita la aplicación de la técnica HOT (ATL satisface esta necesidad), es necesario tener cubiertos los siguientes aspectos:

- **Disponer de un metamodelo que posibilite formular como modelo la información contenida en cualquier W3C-Schema**.

  Tal metamodelo sería un modelo de la especificación *metaschema* de W3C (capa M3), de ahí su posición simbólica en la Figura 3.61 en la parte más alta de M2, casi rozando M3. Análogamente, como los modelos conformes a él representan W3C-Schemas (artefactos en M2), en la figura se sitúan en la parte más alta de M1, casi rozando M2.

- **Disponer de un metamodelo que dé soporte a los modelos de *mappings***. Éste desempeñaría el rol de metamodelo de instrucción, pues los modelos instancia suya instruyen a la metaherramienta.

Metamodelo XSD

EMF proporciona en forma de metamodelo la especificación metaschema que rige la construcción de W3C-Schemas. Se trata del metamodelo `XSD.ecore`. Por motivos de excesiva extensión, no se ilustra aquí el metamodelo en su completitud. La Figura 3.62 muestra su núcleo mientras que la Figura 3.63 y la Figura 3.64 complementan a la primera detallando respectivamente la formulación de tipos y su contenido. El contenedor principal de un modelo XSD es una instancia `XSDSchema` que contiene a través de la asociación-composición `contents` el conjunto de objetos que componen los contenidos de cada W3C-Schema modelado.

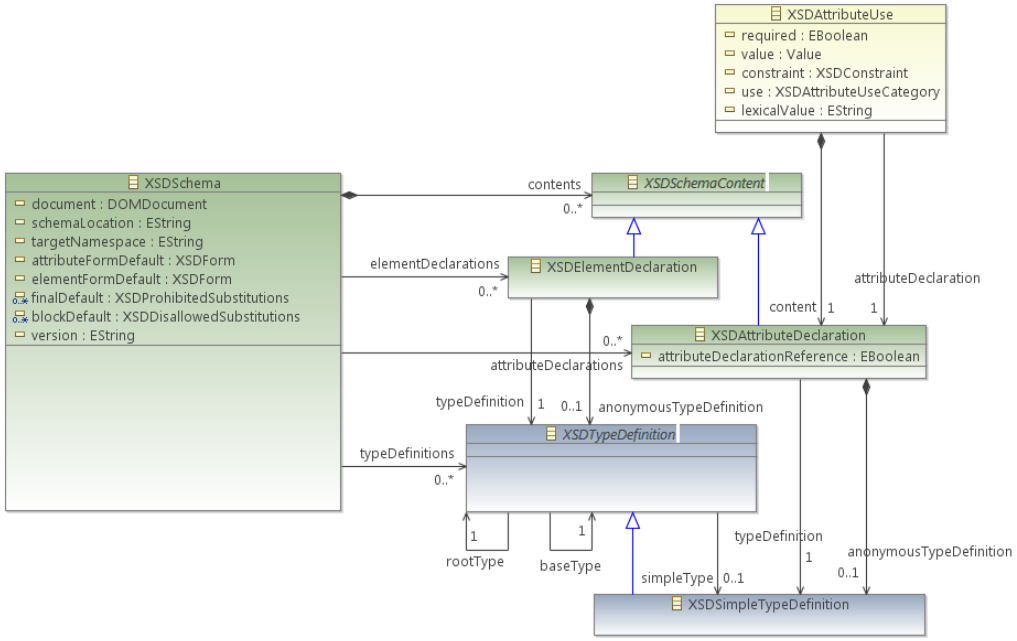


Figura 3.62 – Núcleo del metamodelo `XSD.ecore`

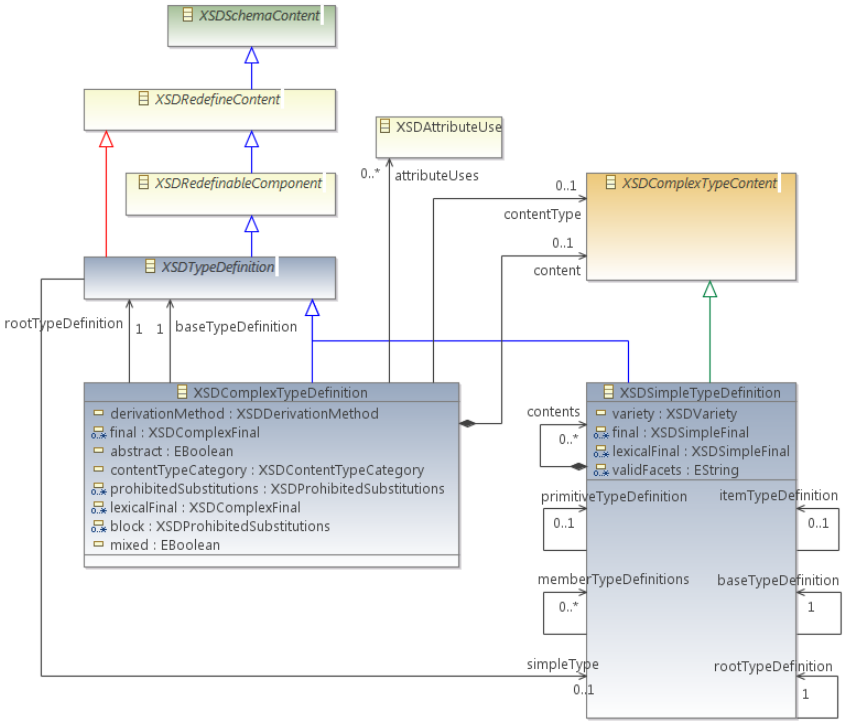


Figura 3.63 – Soporte al modelado de tipos simples y complejos en `XSD.ecore`

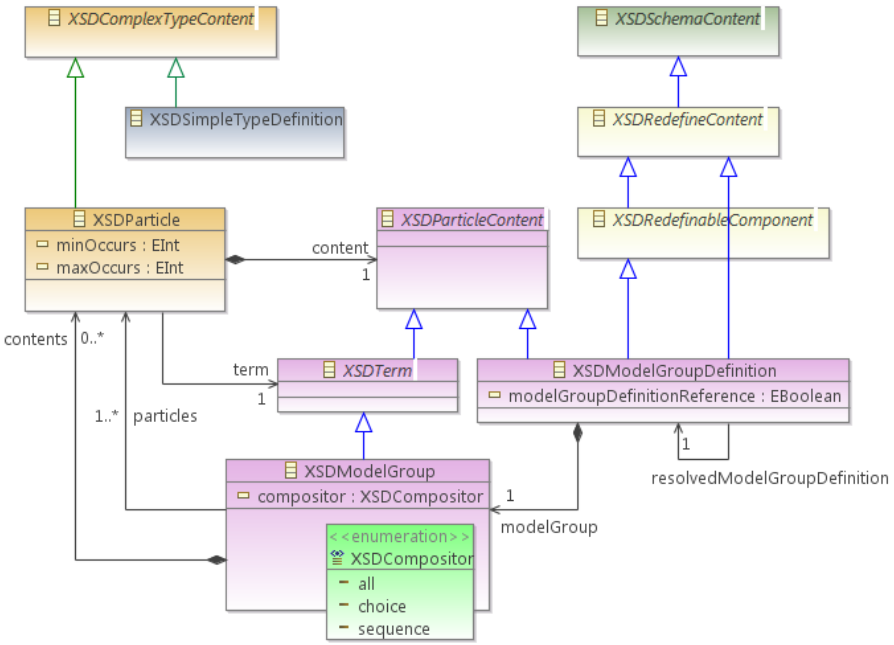


Figura 3.64 – Soporte al modelado de los contenidos de un tipo complejo en `XSD.ecore`

EMF permite obtener la formulación de cualquier W3C-Schema en forma de modelo conforme a `XSD.ecore`, tal y como muestra la Figura 3.65.

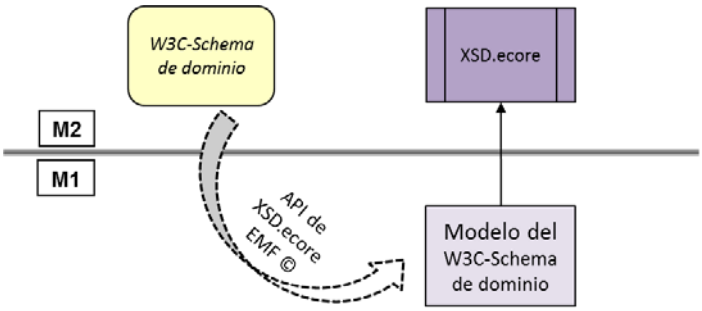


Figura 3.65 – Obtención del modelo XSD correspondiente a un W3C-Schema dado

Metamodelo para mappings

En lugar de diseñar un metamodelo *ad-hoc*, se ha preferido aprovechar el metamodelo `Mapping.ecore` proporcionado por EMF. Éste es adecuado para la formulación de un conjunto genérico de correspondencias en forma de modelo conforme a él.

La elección de este metamodelo para dar soporte a modelos de *mappings* viene respaldada por el hecho de que, durante la creación de un proyecto EMF a partir de un W3C-Schema, el asistente ofrece la posibilidad de generar un modelo de *mappings* que relaciona el contenido del artefacto de partida con el modelo Ecore derivado. Pues bien, tal modelo opcional es conforme a `Mapping.ecore` y además, su generación supone el modelado implícito del W3C-Schema de partida, según el metamodelo `XSD.ecore` presentado en el apartado anterior.

La Figura 3.66 muestra el núcleo del metamodelo `Mapping.ecore`. Tal y como puede observarse, las correspondencias se establecen entre instancias de `EObject`, por lo que cubre correspondencias completamente genéricas. Esto significa que los elementos relacionados

mediante los *mappings* modelados no sólo pueden ser elementos de modelos en M1 sino que, por ejemplo, también puede tratarse de elementos de un metamodelo (clases, atributos, etc.).

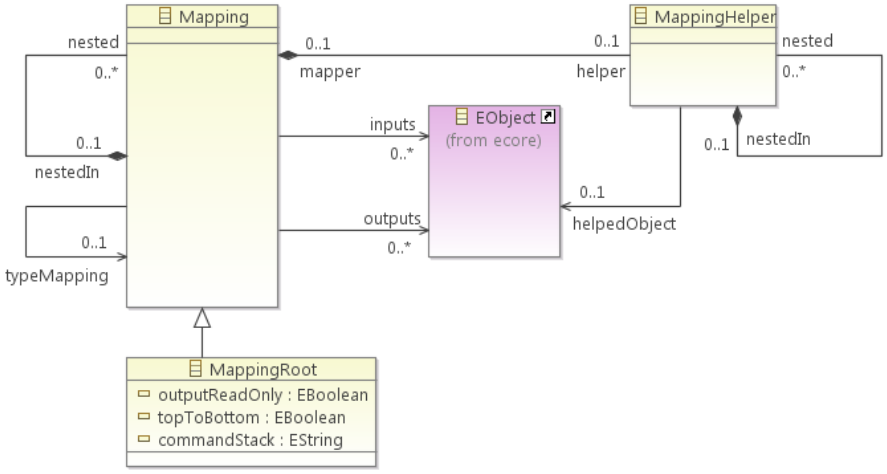


Figura 3.66 – Núcleo del metamodelo `Mapping.ecore`

El contenedor principal de un modelo conforme a `Mapping.ecore` es una instancia `MappingRoot` que recursivamente contiene a través de la asociación-composición `nested` el resto de *mappings* en diversos niveles de anidamiento.

HOTs

Sobre la base de los metamodelos presentados se han de desarrollar las dos HOTs que componen la parte nuclear de la metaherramienta de interoperabilidad XML ↔ Modelware (ver Figura 3.61). Un juego de HOTs tendrá vigencia para una determinada cobertura de estrategias de formalización a nivel de W3C-Schema, de forma que dichas HOTs sepan procesar los modelos de *mappings* entre elementos de metamodelado y construcciones W3C-Schema.

En el siguiente apartado se expone el conjunto de estrategias de formalización a las que se ha considerado dar cobertura. En cualquier caso, la metodología desarrollada queda perfectamente abierta para ser extendida de cara a cubrir más estrategias de formalización.

Estrategias de formalización cubiertas

Se encuentran cubiertas diversas estrategias de formalización a nivel de W3C-Schema, correspondientes a los siguientes elementos de metamodelado[58]:

- Clase concreta.
- Atributo con multiplicidad simple.
- Atributo con multiplicidad múltiple.
- Referencia CON contención y multiplicidad simple.
- Referencia CON contención y multiplicidad múltiple.
- Referencia SIN contención y multiplicidad simple.
- Referencia SIN contención y multiplicidad múltiple.

[58] Multiplicidad simple se refiere a [1] o [0..1] mientras que múltiple a [1..*] o [0..*].

0. **Clase concreta.**

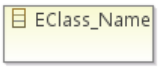


Figura 3.67 – Clase concreta

Para los conceptos de dominio (representados en el metamodelo mediante clases concretas), se cubren las siguientes estrategias de formalización a nivel de W3C-Schema:

a) Un `xs:complexType` (típicamente, su nombre coincide con **`EClass_Name`**).

```
<xs:complexType name="ComplexTypeName">
   ...
</xs:complexType>
```

Código 3.14 – Estrategia de formalización para una clase concreta

1. **Atributo con multiplicidad simple.**

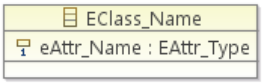


Figura 3.68 – Atributo con multiplicidad simple

Para atributos así caracterizados en las clases del metamodelo de dominio, se cubren las siguientes estrategias de formalización a nivel de W3C-Schema:

a) Un `xs:attribute` dentro del `xs:complexType` correspondiente

- Típicamente, su nombre coincide con **`eAttr_Name.`**
- Su tipo es el `xs:simpleType` correspondiente al tipo de datos **`EAttr_Type.`**
- Requerido u opcional según corresponda.

```
<xs:complexType name="ComplexTypeName">
   ...
   <xs:attribute name="AttrName"
                 type="prefix:SimpleTypeName"
                 use="required" / "optional"/>
   ...
</xs:complexType>
```

Código 3.15 – Estrategia de formalización 1-a

b) Un `xs:element` (de tipo simple) dentro del `xs:complexType` correspondiente.

- Típicamente, su nombre coincide con **`eAttr_Name.`**
- Su tipo es el mismo `xs:simpleType` de la opción a).
- Valor del atributo `minOccurs` según corresponda.

```
<xs:complexType name="ComplexTypeName">
   <xs:sequence>
      ...
      <xs:element name="ElemName"
                  type="prefix:SimpleTypeName"
                  minOccurs="0" / "1"
                  maxOccurs="1"/>
      ...
   <xs:sequence>
</xs:complexType>
```

Código 3.16 – Estrategia de formalización 1-b

c) Un `xs:element` (de tipo complejo) dentro del `xs:complexType` correspondiente.

- Típicamente, su nombre coincide con **`eAttr_Name.`**
- Su tipo es un `xs:complexType` definido a partir del `xs:simpleType` de las opciones anteriores.
- Valor del atributo `minOccurs` según corresponda.

```
<xs:complexType name="ComplexTypeName">
   <xs:sequence>
      ...
      <xs:element name="ElemName"
                  type="prefix:DerivedComplexTypeName"
                  minOccurs="0" / "1"
                  maxOccurs="1"/>
      ...
   <xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="SimpleTypeName_Value">
   <xs:attribute name="Value" type="prefix:SimpleTypeName"/>
</xs:commplexType>
```

Código 3.17 – Estrategia de formalización 1-c

2. **Atributo con multiplicidad múltiple.**

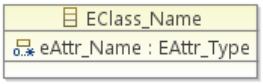


Figura 3.69 – Atributo con multiplicidad múltiple

Para atributos así caracterizados en las clases del metamodelo de dominio, se cubren las siguientes estrategias de formalización a nivel de W3C-Schema:

a) Un `xs:attribute` dentro del `xs:complexType` correspondiente.

- Típicamente, su nombre coincide con **`eAttr_Name.`**
- Su tipo es un `xs:simpleType` definido a partir del `xs:simpleType` correspondiente al tipo de datos **`EAttr_Type`**.
- Requerido u opcional según corresponda.

```
<xs:complexType name="ComplexTypeName">
   ...
   <xs:attribute name="AttrName"
                 type="prefix:DerivedSimpleTypeName"
                 use="required" / "optional"/>
   ...
</xs:complexType>
```

```
<xs:simpleType name="SimpleTypeName_List">

</xs:simpleType>
```

Código 3.18 – Estrategia de formalización 2-a

b) Un `xs:element` (de tipo simple) dentro del `xs:complexType` correspondiente.

- Típicamente, su nombre coincide con **`eAttr_Name.`**
- Su tipo es el mismo `xs:simpleType` de la opción a).
- Valor del atributo `minOccurs` según corresponda.

```
<xs:complexType name="ComplexTypeName">
   <xs:sequence>
      ...
      <xs:element name="Elem_Name"
                  type="prefix:DerivedSimpleTypeName"
                  minOccurs="0" / "1"
                  maxOccurs="1"/>
      ...
   <xs:sequence>
</xs:complexType>
```

Código 3.19 – Estrategia de formalización 2-b

c) Un `xs:element` (de tipo complejo) dentro del `xs:complexType` correspondiente.

- Típicamente, su nombre coincide con **`eAttr_Name`.**
- Su tipo es un `xs:complexType` definido a partir del `xs:simpleType` de las opciones anteriores.
- Valor del atributo `minOccurs` según corresponda.

```
<xs:complexType name="ComplexTypeName">
   <xs:sequence>
      ...
      <xs:element name="ElemName"
                  type="prefix:DerivedComplexTypeName"
                  minOccurs="0" / "1"
                  maxOccurs="1"/>
      ...
   <xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="SimpleTypeName_List_Value">
   <xs:attribute name="Value" type="prefix:SimpleTypeName_List"/>
</xs:complexType>
```

Código 3.20 – Estrategia de formalización 2-c

d) Un `xs:element` (de tipo simple) dentro del `xs:complexType` correspondiente.

Caso análogo a 1-b) con el establecimiento de la faceta `maxOccurs` a infinito.

```
<xs:complexType name="ComplexTypeName">
   <xs:sequence>
      ...
      <xs:element name="ElemName"
                  type="prefix:SimpleTypeName"
                  minOccurs="0" / "1"
                  maxOccurs="unbounded"/>
      ...
   <xs:sequence>
</xs:complexType>
```

Código 3.21 – Estrategia de formalización 2-d

e) Un `xs:element` (de tipo complejo) dentro del `xs:complexType` correspondiente.

Caso análogo a 1-c) con el establecimiento de la faceta `maxOccurs` a infinito.

```
<xs:complexType name="ComplexTypeName">
   <xs:sequence>
      ...
      <xs:element name="ElemName"
                  type="prefix:ComplexTypeName"
                  minOccurs="0" / "1"
                  maxOccurs="unbounded"/>
      ...
   <xs:sequence>
</xs:complexType>
```

Código 3.22 – Estrategia de formalización 2-e

3. **Referencia CON contención y multiplicidad simple.**

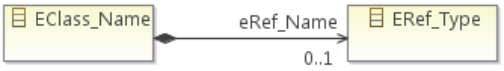


Figura 3.70 – Referencia con contención y multiplicidad simple

Para referencias así caracterizadas en las clases del metamodelo de dominio, se cubren las siguientes estrategias de formalización a nivel de W3C-Schema:

a) **En el caso más genérico**, una `xs:choice`.

- El tipo de cada `xs:element` es uno de los `xs:complexType` correspondientes a las subclases concretas en que se desglosa el tipo de la referencia inicial.
- Requerida u opcional según corresponda.

```
<xs:complexType name="ComplexTypeName">
   <xs:choice (minOccurs="0")>
      <xs:element name="ComplexType1Name"
                  type="prefix:ComplexType1Name"/>
      ...
      ...
      <xs:element name="ComplexTypeNName"
                  type="prefix:ComplexTypeNName"/>
   </xs:choice>
</xs:complexType>
```

Código 3.23 – Estrategia de formalización 3-a

b) **En el caso particular de que no haya desglose en subclases**, un único `xs:element` dentro del `xs:complexType` correspondiente.

- Típicamente, **`ElemName`** coincide con **`eRef_Name.`**
- Su tipo es el `xs:complexType` correspondiente a la clase que tipa a la referencia inicial.
- Requerido u opcional según corresponda.

```
<xs:complexType name="ComplexTypeName">
   <xs:sequence>
      ...
      <xs:element name="ElemName"
                  type="prefix:TargetComplexTypeName"
                  (minOccurs="0")/>
      ...
   </xs:sequence>
</xs:complexType>
```

Código 3.24 – Estrategia de formalización 3-b

4. **Referencia CON contención y multiplicidad múltiple.**

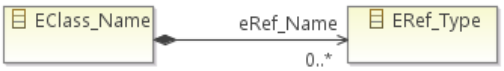


Figura 3.71 – Referencia con contención y multiplicidad múltiple

Caso análogo al anterior, tanto en la situación más genérica como en su variante más particular, pero ahora con el establecimiento explícito de la faceta `maxOccurs` a infinito, bien en la `xs:choice` o bien en el único `xs:element`.

5. **Referencia SIN contención y multiplicidad simple.**

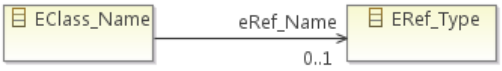


Figura 3.72 – Referencia sin contención y multiplicidad simple

Para referencias así caracterizadas en las clases del metamodelo de dominio, se cubren las siguientes estrategias de formalización a nivel de W3C-Schema:

a) Un `xs:attribute` dentro del `xs:complexType` correspondiente.

- Típicamente, su nombre coincide con **`eRef_Name`.**
- Su tipo es el `xs:simpleType` empleado a efectos identificativos.
- Requerido u opcional según corresponda.

```
<xs:complexType name="ComplexTypeName">
   ...
   <xs:attribute name="AttrName"
                 type="prefix:ID_SimpleTypeName"
                 use="required" / "optional"/>
   ...
</xs:complexType>
```

Código 3.25 – Estrategia de formalización 5-a

b) Un `xs:element` (de tipo simple) dentro del `xs:complexType` correspondiente.

- Típicamente, su nombre coincide con **`eRef_Name`.**
- Su tipo es el mismo `xs:simpleType` de la opción a).
- Con el valor del atributo `minOccurs` según corresponda.

```
<xs:complexType name="ComplexTypeName">
   <xs:sequence>
      ...
      <xs:element name="ElemName"
                  type="prefix:ID_SimpleTypeName"
                  minOccurs="0" / "1"
                  maxOccurs="1"/>
      ...
   <xs:sequence>
</xs:complexType>
```

Código 3.26 – Estrategia de formalización 5-b

c) Un `xs:element` (de tipo complejo) dentro del `xs:complexType` correspondiente.

- Típicamente, su nombre coincide con **`eRef_Name`.**
- Su tipo es un `xs:complexType` definido a partir del `xs:simpleType` de las opciones anteriores.
- Con el valor del atributo `minOccurs` según corresponda.

```
<xs:complexType name="ComplexTypeName">
   <xs:sequence>
      ...
      <xs:element name="Elem_Name"
                  type="prefix:DerivedComplexTypeName"
                  minOccurs="0" / "1"
                  maxOccurs="1"/>
      ...
   <xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="ID_SimpleTypeName_Value">
   <xs:attribute name="Value" type="prefix:ID_SimpleTypeName"/>
</xs:commplexType>
```

Código 3.27 – Estrategia de formalización 5-c

6. **Referencia SIN contención y multiplicidad múltiple.**

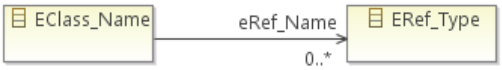


Figura 3.73 – Referencia sin contención y multiplicidad múltiple

Para referencias así caracterizadas en las clases del metamodelo de dominio, se cubren las siguientes estrategias de formalización a nivel de W3C-Schema:

a) Un `xs:attribute` dentro del `xs:complexType` correspondiente.

- Típicamente, su nombre coincide con **`eRef_Name.`**
- Su tipo es un `xs:simpleType` definido a partir del `xs:simpleType` empleado a efectos identificativos.
- Requerido u opcional según corresponda.

```
<xs:complexType name="ComplexTypeName">
   ...
   <xs:attribute name="AttrName"
                 type="prefix:DerivedSimpleTypeName"
                 use="required" / "optional"/>
   ...
</xs:complexType>
```

```
<xs:simpleType name="ID_SimpleTypeName_List">

</xs:simpleType>
```

Código 3.28 – Estrategia de formalización 6-a

b) Un `xs:element` (de tipo simple) dentro del `xs:complexType` correspondiente.

- Típicamente, su nombre coincide con **`eRef_Name.`**
- Su tipo es el mismo `xs:simpleType` de la opción a).
- Valor del atributo `minOccurs` según corresponda.

```
<xs:complexType name="ComplexTypeName">
   <xs:sequence>
      ...
      <xs:element name="ElemName"
                  type="prefix:DerivedSimpleTypeName"
                  minOccurs="0" / "1"
                  maxOccurs="1"/>
      ...
   <xs:sequence>
</xs:complexType>
```

Código 3.29 – Estrategia de formalización 6-b

c) Un `xs:element` (de tipo complejo) dentro del `xs:complexType` correspondiente.

- Típicamente, su nombre coincide con **`eRef_Name.`**

- Su tipo es un `xs:complexType` definido a partir del `xs:simpleType` de las opciones anteriores.
- Valor del atributo `minOccurs` según corresponda.

```
<xs:complexType name="ComplexTypeName">
   <xs:sequence>
      …
      <xs:element name="ElemName"
                  type="prefix:DerivedComplexTypeName"
                  minOccurs="0" / "1"
                  maxOccurs="1"/>
      …
   <xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="ID_SimpleTypeName_List_Value">
   <xs:attribute name="Value" type="prefix:ID_SimpleTypeName_List"/>
</xs:complexType>
```

Código 3.30 – Estrategia de formalización 6-c

d) Un `xs:element` (de tipo simple) dentro del `xs:complexType` correspondiente.

Caso análogo a 5-b) con el establecimiento de la faceta `maxOccurs` a infinito.

```
<xs:complexType name="ComplexTypeName">
   <xs:sequence>
      ...
      <xs:element name="ElemName"
                  type="prefix:ID_SimpleTypeName"
                  minOccurs="0" / "1"
                  maxOccurs="unbounded"/>
      ...
   <xs:sequence>
</xs:complexType>
```

Código 3.31 – Estrategia de formalización 6-d

e) Un `xs:element` (de tipo complejo) dentro del `xs:complexType` correspondiente.

Caso análogo a 5-c) con el establecimiento de la faceta `maxOccurs` a infinito.

```
<xs:complexType name="ComplexTypeName">
   <xs:sequence>
      ...
      <xs:element name="ElemName"
                  type="prefix:DerivedComplexTypeName"
                  minOccurs="0" / "1"
                  maxOccurs="unbounded"/>
      ...
   <xs:sequence>
</xs:complexType>
```

Código 3.32 – Estrategia de formalización 6-e

Estructura de las transformaciones XML_to_MM

El Código 3.33 presenta la estructura de la implementación ATL de las transformaciones *XML_to_MM* generadas por la primera HOT.

```
-- @nsURI XML=http://www.eclipse.org/am3/2007/XML
-- @path MM=...

-- Header section
module XML_to_MM;
create OUTmodel : MM from INmodel : XML;

-- ************************ ATTRIBUTE helpers ********************************
```

```
-- ************************ Functional HELPERS ******************************
helper context XML!Element def : getTextNode() : String = ...
helper context XML!Element def : hasAttributes() : Boolean = ...
helper context XML!Element def : getAttributes() : Sequence(XML!Attribute) = ...
helper context XML!Element def : hasAttribute(attrName : String) : Boolean = ...
helper context XML!Element def : getStringAttrValue(attrName : String) : String = ...
helper context XML!Element def : getIntAttrValue(attrName : String) : Integer = ...
helper context XML!Element def : getRealAttrValue(attrName : String) : Real = ...
helper context XML!Element def : getBoolAttrValue(attrName : String) : Boolean = ...
helper context XML!Element def : hasChildrenElem() : Boolean = ...
helper context XML!Element def : hasChildrenElemWithName(name : String) : Boolean = ...
helper context XML!Element def : getChildrenElem() : Sequence(XML!Element) = ...
helper context XML!Element def : getChildrenElemWithName(name : String) :
                                                        Sequence(XML!Element) = ...
-- ************************ CALLED RULES ************************************


-- ************************ MATCHED RULES ***********************************
rule Root_Elem {...}

rule ComplexType1_Name_to_EClass1_Name {...}
...
rule ComplexTypeN_Name_to_EClassN_Name {...}
```

Código 3.33 – Estructura del código ATL de las transformaciones XML_to_MM

Como se observa en el Código 3.33, no aparecen ni atributos ni reglas invocadas, por lo que únicamente se identifican dos bloques:

- **Bloque de *helpers***. Está formado por *helpers* para gestión básica de instancias `Element` y dada su signatura autodescriptiva no se entra aquí en mayor detalle. Se trata de un bloque siempre idéntico en cualquier transformación *XML_to_MM*, por lo que podría convertirse en una librería ATL.
- **Bloque de reglas *matched***. Incluye una regla por cada correspondencia entre clase concreta y tipo complejo. La estructura de cada regla es claramente regular, tal y como muestra el Código 3.34.

```
rule ComplexTypeName_to_EClass_Name {
   from
      input : XML!Element (input.name = 'prefix:ComplexTypeName')
   to
      output : MM!EClass_Name (
      -- attrs.
         EAttr1_Name <- binding for EAttr1,
         ...
         EAttrN_Name <- binding for EAttrN,
      -- refs
         ContainmentERef1_Name <- binding for ContainmentERef1,
         ...
         ContainmentERefN_Name <- binding for ContainmentERefN
      )
   do {
      output.ERef1_Name <- binding statement for ERef1,
      ...
      output.ERefN_Name <- binding statement for ERefN
   }
}
```

Código 3.34 – Estructura de las reglas *matched* en las transformaciones XML_to_MM

Patrones en la implementación de los bindings

El código que implementa cada *binding* depende de la estrategia de formalización empleada en el correspondiente W3C-Schema para los atributos y referencias en cuestión. Así, para las estrategias de formalización consideradas, se siguen los siguientes patrones en la implementación de los *bindings*.

1. **Atributo de multiplicidad simple**.

a) Formalizado según la estrategia 1-a.

La implementación mostrada en el Código 3.35 si se trata de un atributo opcional o simplemente el código de asignación, sin necesidad de la estructura ***if*** declarativa, si se trata de un atributo obligatorio.

```
EAttr_Name <- if input.hasAttribute('AttrName') then
                input.getValueOf{String,Int,Real,Bool}Attr('AttrName')
              else
                OclUndefined
              endif
```

Código 3.35 – Patrón de *binding* para atributo formalizado según la estrategia 1-a

b) Formalizado según la estrategia 1-b.

La implementación mostrada en el Código 3.36 si se trata de un atributo opcional o simplemente el código de asignación, sin necesidad de la estructura ***if*** declarativa, si se trata de un atributo obligatorio.

```
EAttr_Name <- if input.hasChildrenElemWithName('prefix:ElemName') then
              (input.getChildrenElemWithName('prefix:ElemName') -> first()).getTextNode()
              else
                OclUndefined
              endif
```

Código 3.36 – Patrón de *binding* para atributo formalizado según la estrategia 1-b

c) Formalizado según la estrategia 1-c.

La implementación mostrada en el Código 3.37 si se trata de un atributo opcional o simplemente el código de asignación, sin necesidad de la estructura ***if*** declarativa, si se trata de un atributo obligatorio.

```
EAttr_Name <- if input.hasChildrenElemWithName('prefix:ElemName') then
                (input.getChildrenElemWithName('prefix:ElemName') -> first()).
                                          getValueOf{String,Int,Real,Bool}Attr('Value')
              else
                OclUndefined
              endif
```

Código 3.37 – Patrón de *binding* para atributo formalizado según la estrategia 1-c

2. **Atributo de multiplicidad múltiple**.

a) Formalizado según la estrategia 2-a.

La implementación mostrada en el Código 3.38 si se trata de un atributo [0..*] o simplemente el código de asignación, sin necesidad de la estructura ***if*** declarativa, si es un atributo [1..*].

```
EAttr_Name <- if input.hasAttribute('AttrName') then
                input.getValueOfStringAttr('AttrName').split(' '))
              else
                 OclUndefined
              endif
```

Código 3.38 – Patrón de *binding* para atributo formalizado según la estrategia 2-a

b) Formalizado según la estrategia 2-b.

La implementación mostrada en el Código 3.39 si se trata de un atributo [0..*] o simplemente el código de asignación, sin necesidad de la estructura ***if*** declarativa, si es un atributo [1..*].

```
EAttr_Name <- if input.hasChildrenElemWithName('prefix:ElemName') then
                (input.getChildrenElemWithName('prefix:ElemName') -> first()).
                                                  getTextNode().split(' ')
              else
                 OclUndefined
              endif
```

Código 3.39 – Patrón de *binding* para atributo formalizado según la estrategia 2-b

c) Formalizado según la estrategia 2-c.

La implementación mostrada en el Código 3.40 si se trata de un atributo [0..*] o simplemente el código de asignación, sin necesidad de la estructura ***if*** declarativa, si se trata de un atributo [1..*].

```
EAttr_Name <- if input.hasChildrenElemWithName('prefix:ElemName') then
                (input.getChildrenElemWithName('prefix:ElemName') -> first()).
                                  getValueOfStringAttr('Value').split(' ')
              else
                 OclUndefined
              endif
```

Código 3.40 – Patrón de *binding* para atributo formalizado según la estrategia 2-c

d) Formalizado según la estrategia 2-d.

La implementación mostrada en el Código 3.41 si se trata de un atributo [0..*] o simplemente el código de asignación, sin necesidad de la estructura ***if*** declarativa, si se trata de un atributo [1..*].

```
EAttr_Name <- if input.hasChildrenElemWithName('prefix:ElemName') then
                input.getChildrenElemWithName('prefix:ElemName') ->
                collect(e | e.getTextNode())
              else
                 OclUndefined
              endif
```

Código 3.41 – Patrón de *binding* para atributo formalizado según la estrategia 2-d

e) Formalizado según la estrategia 2-e.

La implementación mostrada en el Código 3.42 si se trata de un atributo [0..*] o simplemente el código de asignación, sin necesidad de la estructura ***if*** declarativa, si se trata de un atributo [1..*].

```
EAttr_Name <- if input.hasChildrenElemWithName('prefix:ElemName') then
                input.getChildrenElemWithName('prefix:ElemName') ->
                collect(e | e.getValueOfStringAttr('Value'))
              else
                 OclUndefined
              endif
```

Código 3.42 – Patrón de *binding* para atributo formalizado según la estrategia 2-e

3. **Referencia CON contención de multiplicidad simple**.

Formalizada según las estrategias 3-a y 3-b.

La implementación mostrada en el Código 3.43 si se trata de una referencia opcional o simplemente el código de asignación, sin necesidad de la estructura ***if*** declarativa, si se trata de una referencia obligatoria.

```
ERef_Name <- if input.hasChildrenElem() then
                input.getChildrenElem() ->
                    select(e | e.name = 'ComplexType1_QualifiedName' or
                               e.name = 'ComplexType2_QualifiedName' or
                               ...
                               e.name = 'ComplexTypeN_QualifiedName') -> first()
             else
                  OclUndefined
             endif
```

Código 3.43 – Patrón de *binding* para referencia formalizada según las estrategias 3-a y 3-b

4. **Referencia CON contención de multiplicidad múltiple**.

   Caso análogo al anterior pero sin necesidad de la invocación final "*-> first()* ".

Patrones en la implementación de los binding statements

El código que implementa cada *binding statement* también depende de la estrategia de formalización empleada en el correspondiente W3C-Schema para las referencias en cuestión. Así, para las estrategias de formalización consideradas, se siguen los siguientes patrones en la implementación de los *binding statement*.

1. **Referencia SIN contención de multiplicidad simple**.

a) Formalizada según la estrategia 5-a.

   La implementación mostrada en el Código 3.44 si se trata de una referencia opcional o simplemente el código de asignación, sin necesidad de la estructura ***if*** imperativa, si se trata de una referencia obligatoria.

```
if (input.hasAttribute('AttrName')) {
   output.ERef_Name <- MM!ERef_Type.allInstances() ->
                       select(e | e.idAttr = input.getValueOfStringAttr('AttrName')) ->
                                                                                first();
}
```

Código 3.44 – Patrón de *binding statement* para referencia formalizada según la estrategia 5-a

b) Formalizada según la estrategia 5-b.

   La implementación mostrada en el Código 3.45 si se trata de una referencia opcional o simplemente el código de asignación, sin necesidad de la estructura ***if*** imperativa, si se trata de una referencia obligatoria.

```
if (input.hasChildrenElemWithName('prefix:ElemName')) {
   output.ERef_Name <- MM!ERef_Type.allInstances() ->
                select(e | e.idAttr = (input.getChildrenElemWithName('prefix:ElemName') ->
                                                first()).getTextNode()) -> first();
}
```

Código 3.45 – Patrón de *binding statement* para referencia formalizada según la estrategia 5-b

c) Formalizada según la estrategia 5-c.

   La implementación mostrada en el Código 3.46 si se trata de una referencia opcional o simplemente el código de asignación, sin necesidad de la estructura ***if*** imperativa, si se trata de una referencia obligatoria.

```
if (input.hasChildrenElemWithName('prefix:ElemName')) {
   output.ERef_Name <- MM!ERef_Type.allInstances() ->
               select(e | e.idAttr = (input.getChildrenElemWithName('prefix:ElemName') ->
                                          first()).getValueOfStringAttr('Value')) ->
                                                                                  first();
}
```

Código 3.46 – Patrón de *binding statement* para referencia formalizada según la estrategia 5-c

2. **Referencia SIN contención de multiplicidad múltiple**.

a) Formalizada según la estrategia 6-a.

La implementación mostrada en el Código 3.47 si se trata de una referencia [0..*] o simplemente el bucle ***for***, sin necesidad de la estructura ***if*** imperativa, si se trata de una referencia [1..*].

```
if (input.hasAttribute('AttrName')) {
   for (token in input.getValueOfStringAttr('AttrName').split(' ')) {
      output.ERef_Name <- MM!ERef_Type.allInstances() ->
                          select(e | e.idAttr = token) -> first();
    }
}
```

Código 3.47 – Patrón de *binding statement* para referencia formalizada según la estrategia 6-a

b) Formalizada según la estrategia 6-b.

La implementación mostrada en el Código 3.48 si se trata de una referencia [0..*] o simplemente el bucle ***for***, sin necesidad de la estructura ***if*** imperativa, si se trata de una referencia [1..*].

```
if (input.hasChildrenElemWithName('prefix:ElemName')) {
   for (token in (input.getChildrenElemWithName('prefix:ElemName') -> first()).
                                                          getTextNode().split(' ')) {
      output.ERef_Name <- MM!ERef_Type.allInstances() ->
                          select(e | e.idAttr = token) -> first();
    }
}
```

Código 3.48 – Patrón de *binding statement* para referencia formalizada según la estrategia 6-b

c) Formalizada según la estrategia 6-c.

La implementación mostrada en el Código 3.49 si se trata de una referencia [0..*] o simplemente el bucle ***for***, sin necesidad de la estructura ***if*** imperativa, si se trata de una referencia [1..*].

```
if (input.hasChildrenElemWithName('prefix:ElemName')) {
   for (token in (input.getChildrenElemWithName('prefix:ElemName') -> first()).
                                          getValueOfStringAttr('Value').split(' ')) {
      output.ERef_Name <- MM!ERef_Type.allInstances() ->
                          select(e | e.idAttr = token) -> first();
    }
}
```

Código 3.49 – Patrón de *binding statement* para referencia formalizada según la estrategia 6-c

d) Formalizada según la estrategia 6-d.

La implementación mostrada en el Código 3.50 si se trata de una referencia [0..*] o simplemente el bucle ***for***, sin necesidad de la estructura ***if*** imperativa, si se trata de una referencia [1..*].

```
if (input.hasChildrenElemWithName('prefix:ElemName')) {
   for (elem in input.getChildrenElemWithName('prefix:ElemName')) {
      output.ERef_Name <- MM!ERef_Type.allInstances() ->
                          select(e | e.idAttr = elem.getTextNode()) -> first();
    }
}
```

Código 3.50 – Patrón de *binding statement* para referencia formalizada según la estrategia 6-d

e) Formalizada según la estrategia 6-e.

La implementación mostrada en el Código 3.51 si se trata de una referencia [0..*] o simplemente el bucle ***for***, sin necesidad de la estructura ***if*** imperativa, si se trata de una referencia [1..*].

```
if (input.hasChildrenElemWithName('prefix:ElemName')) {
   for (elem in input.getChildrenElemWithName('prefix:ElemName')) {
      output.ERef_Name <- MM!ERef_Type.allInstances() ->
                          select(e | e.idAttr = elem.getValueOfStringAttr('Value')) ->
                          first();
    }
}
```

Código 3.51 – Patrón de *binding statement* para referencia formalizada según la estrategia 6-e

Estructura de las transformaciones MM_to_XML

El Código 3.52 presenta la estructura de la implementación ATL de las transformaciones *MM_to_XML* generadas por la segunda HOT.

```
-- @path MM=...
-- @nsURI XML=http://www.eclipse.org/am3/2007/XML

-- Header section
module MM_to_XML;
create OUTmodel : XML from INmodel : MM;


-- ************************ ATTRIBUTE helpers ******************************


-- ************************ Functional HELPERS *****************************


-- ************************ CALLED RULES ***********************************
rule CreateXMLattr_StringValue(attrName : String,
                               attrValue : String,
                               attrParent : XML!Element) {...}

rule ContributeXMLelem_OfType_Model_Elem_Ref (localName : String,
                                              parent : XML!Element,
                                              modelElemName : String) {...}

-- ************************ MATCHED RULES **********************************
rule Mast_Model {...}

rule EClass1_Name_to_ComplexType1_Name {...}
...
...
rule EClassN_Name_to_ComplexTypeN_Name {...}
```

Código 3.52 – Estructura del código ATL de las transformaciones MM_to_XML

Como se observa en el Código 3.52, no aparecen atributos ni *helpers*, por lo que únicamente se identifican dos bloques:

- **Bloque de reglas invocadas**.

- **Bloque de reglas *matched***. Incluye una regla por cada correspondencia entre clase concreta y tipo complejo. La estructura de cada regla es claramente regular, tal y como muestra el Código 3.53.

```
rule EClass_Name_to_ComplexTypeName {
   from
      input : MM!EClass_Name
   using {

   }
   to
      output : XML!Element (
         name <- 'prefix:ComplexTypeName'
--       parent <-
         children <- binding for ContainmentERef_1,
         ...
         children <- binding for ContainmentERef_N,
   )
   do {
      called rule invocation for EAttr1,
      ...
      called rule invocation for EAttrN
      called rule invocation for ERef1,
      ...
      called rule invocation for ERefN
   }
}
```

Código 3.53 – Estructura de las reglas *matched* en las transformaciones MM_to_XML

Patrones en la implementación de los bindings

El código que implementa cada *binding* es dependiente de la estrategia de formalización empleada en el correspondiente W3C-Schema para las referencias en cuestión. En este caso, se sigue un único patrón en la implementación de los *bindings*.

1. **Referencia CON contención**. Independientemente de su multiplicidad, la implementación mostrada en el Código 3.54.

```
children <- input.ERef_Name
```

Código 3.54 – Patrón de *binding* para referencia formalizada según estrategias 3-a, 3-b, 4-a y 4-b

Patrones en la implementación de los binding statements

El código que implementa cada *binding statement* también depende de la estrategia de formalización empleada en el correspondiente W3C-Schema para los atributos y referencias en cuestión. Así, se siguen los siguientes patrones en la implementación de los *binding statement*.

1. **Atributo de multiplicidad simple**.

a) Formalizado según la estrategia 1-a.

La implementación mostrada en Código 3.55.

```
thisModule.CreateXMLattr_StringValue('AttrName',
                                     input.EAttr_Name.toString(),
                                     output);
```

Código 3.55 – Patrón de *binding statement* para atributo formalizado según la estrategia 1-a

b) Formalizado según la estrategia 1-b.

La implementación mostrada en Código 3.56.

```
thisModule.ContributeXMLelem_TextNode('ElemName',
                                      input.EAttr_Name.toString(),
                                      output);
```

Código 3.56 – Patrón de *binding statement* para atributo formalizado según la estrategia 1-b

c) Formalizado según la estrategia 1-c.

La implementación mostrada en el Código 3.57.

```
thisModule.ContributeXMLelem_AttrValue('ElemName',
                                       input.EAttr_Name.toString(),
                                       output);
```

Código 3.57 – Patrón de *binding statement* para atributo formalizado según la estrategia 1-c

2. **Atributo de multiplicidad múltiple**.

a) Formalizado según la estrategia 2-a.

La implementación mostrada en el Código 3.58.

```
using {
   values : String = '';
}
do {
   for (e in input.EAttr_Name) {
      values <- values + e + ' ';
   }
   thisModule.CreateXMLattr_StringValue('AttrName',
                                        values.trim(),
                                        output);
}
```

Código 3.58 – Patrón de *binding statement* para atributo formalizado según la estrategia 2-a

b) Formalizado según la estrategia 2-b.

La implementación mostrada en el Código 3.59.

```
using {
   values : String = '';
}
do {
   for (e in input.EAttr_Name) {
      values <- values + e + ' ';
   }
   thisModule.ContributeXMLelem_TextNode('ElemName',
                                         values.trim(),
                                         output);
}
```

Código 3.59 – Patrón de *binding statement* para atributo formalizado según la estrategia 2-b

c) Formalizado según la estrategia 2-c.

La implementación mostrada en el Código 3.60.

```
using {
   values : String = '';
}
do {
   for (e in input.EAttr_Name) {
      values <- values + e + ' ';
   }
   thisModule.ContributeXMLelem_AttrValue('ElemName',
                                          values.trim(),
                                          output);
}
```

Código 3.60 – Patrón de *binding statement* para atributo formalizado según la estrategia 2-c

d) Formalizado según la estrategia 2-d.

La implementación mostrada en el Código 3.61.

```
for (e in input.EAttr_Name) {
   thisModule.ContributeXMLelem_TextNode('ElemName',
                                         e,
                                         output);
}
```

Código 3.61 – Patrón de *binding statement* para atributo formalizado según la estrategia 2-d

e) Formalizado según la estrategia 2-e.

La implementación mostrada en el Código 3.62.

```
for (e in input.EAttr_Name) {
   thisModule.ContributeXMLelem_AttrValue('ElemName',
                                          e,
                                          output);
}
```

Código 3.62 – Patrón de *binding statement* para atributo formalizado según la estrategia 2-e

3. **Referencia SIN contención y multiplicidad simple**.

a) Formalizada según la estrategia 5-a.

La implementación mostrada en el Código 3.63 si se trata de una referencia opcional o simplemente el código de invocación de regla sin necesidad de la estructura ***if*** imperativa, si se trata de una referencia obligatoria.

```
if (not input.ERef_Name.oclIsUndefined()) {
   thisModule.CreateXMLattr_StringValue('AttrName',
                                        input.ERef_Name.idAttr,
                                        output);
}
```

Código 3.63 – Patrón de *binding statement* para referencia formalizada según la estrategia 5-a

b) Formalizada según la estrategia 5-b.

La implementación mostrada en el Código 3.64 si se trata de una referencia opcional o simplemente el código de invocación de regla sin necesidad de la estructura ***if*** imperativa, si se trata de una referencia obligatoria.

```
if (not input.ERef_Name.oclIsUndefined()) {
   thisModule.ContributeXMLelem_TextNode('ElemName',
                                         input.ERef_Name.idAttr,
                                         output);
}
```

Código 3.64 – Patrón de *binding statement* para referencia formalizada según la estrategia 5-b

c) Formalizada según la estrategia 5-c.

La implementación mostrada en el Código 3.65 si se trata de una referencia opcional o simplemente el código de invocación de regla sin necesidad de la estructura ***if*** imperativa, si se trata de una referencia obligatoria.

```
if (not input.ERef_Name.oclIsUndefined()) {
   thisModule.CreateXMLelem_AttrValue('ElemName',
                                      input.ERef_Name.idAttr,
                                      output);
}
```

Código 3.65 – Patrón de *binding statement* para referencia formalizada según la estrategia 5-c

4. **Referencia SIN contención y multiplicidad múltiple**.

a) Formalizada según la estrategia 6-a.

La implementación mostrada en el Código 3.66.

```
using {
   ids : String = '';
}
do {
   for (e in input.ERef_Name) {
      ids <- ids + e.idAttr + ' ';
   }
   thisModule.CreateXMLattr_StringValue('AttrName',
                                        ids.trim(),
                                        output);
}
```

Código 3.66 – Patrón de *binding statement* para referencia formalizada según la estrategia 6-a

b) Formalizada según la estrategia 6-b.

La implementación mostrada en el Código 3.67.

```
using {
   ids : String = '';
}
do {
   for (e in input.ERef_Name) {
      ids <- ids + e.idAttr + ' ';
   }
   thisModule.ContributeXMLelem_TextNode('ElemName',
                                         ids.trim(),
                                         output);
}
```

Código 3.67 – Patrón de *binding statement* para referencia formalizada según la estrategia 6-b

c) Formalizada según la estrategia 6-c.

La implementación mostrada en el Código 3.68.

```
using {
   ids : String = '';
}
do {
   for (e in input.ERef_Name) {
      ids <- ids + e.idAttr + ' ';
   }
   thisModule.ContributeXMLelem_AttrValue('ElemName',
                                          ids.trim(),
                                          output);
}
```

Código 3.68 – Patrón de *binding statement* para referencia formalizada según la estrategia 6-c

d) Formalizada según la estrategia 6-d.

La implementación mostrada en el Código 3.69.

```
for (e in input.ERef_Name) {
   thisModule.ContributeXMLelem_TextNode('ElemName',
                                         e.idAttr
                                         output);
}
```

Código 3.69 – Patrón de *binding statement* para referencia formalizada según la estrategia 6-d

e) Formalizada según la estrategia 6-e.

La implementación mostrada en el Código 3.70.

```
for (e in input.ERef_Name) {
     thisModule.ContributeXMLelem_AttrValue('ElemName',
                                            e.idAttr
                                            output);
}
```

Código 3.70 – Patrón de *binding statement* para referencia formalizada según la estrategia 6-e.

La subsección a continuación aborda la aplicación al escenario MAST-2 de la estrategia de interoperabilidad diseñada, comenzando por las correspondencias entre elementos del metamodelo y del W3C-Schema, para posteriormente presentar las respectivas transformaciones M2M de inyección y extracción implementadas en base a tales correspondencias. Después, la subsección 3.4.4 aborda, también en MAST-2, el plano de la automatización.

3.4.3 Interoperabilidad XML ↔ Modelware en MAST-2

El entorno MAST-2 representa un banco de trabajo ideal sobre el que implementar la estrategia diseñada para interoperabilidad XML ↔ Modelware, pues es un dominio que se encuentra formalizado mediante un W3C-Schema (`Mast2_Model.xsd`) y un metamodelo (`Mast2.ecore`) desarrollados de manera independiente. El primero es cronológicamente anterior y existe un conjunto estable de herramientas que operan sobre sus correspondientes documentos XML. Sin embargo, en su diseño heterodoxo y excesivamente laxo no se hizo uso de gran parte de las posibilidades expresivas y de formalización que ofrece la tecnología W3C-Schema. Por tanto, no fue empleado como base para la obtención automática de `Mast2.ecore`, sino que éste fue desarrollado explícita y manualmente, dando así lugar a una desalineación entre ambos artefactos. Así, para poder aplicar las herramientas MAST a modelos conformes a `Mast2.ecore`, se requiere su conversión previa a documento XML según `Mast2_Model.xsd`, la cual es llevada a cabo gracias a la implementación de la estrategia de interoperabilidad diseñada.

Todas las estrategias de formalización empleadas en `Mast2_Model.xsd` se encuentran cubiertas. En concreto se trata de las identificadas en el apartado anterior como: 0, 1a, 2a, 3a, 3b, 4a, 4b, 5a y 6a. El siguiente apartado expone algunos ejemplos de correspondencias entre las formalizaciones a nivel de W3C-Schema y de metamodelo.

Ejemplos de correspondencias

Ejemplo 1. La correspondencia entre la clase `Regular_Processor` (Figura 3.74) y el tipo complejo de igual nombre (Código 3.71) ejemplifica las estrategias 0, 1a, 5a y 6b.

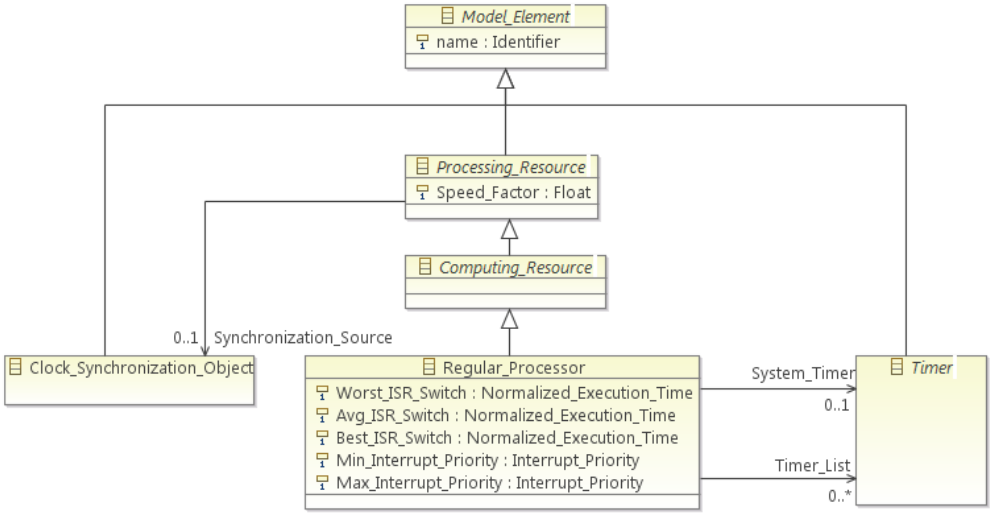


Figura 3.74 – Clase `Regular_Processor`

```
<xs:complexType name="Regular_Processor">

   <xs:sequence>
      <xs:element name="Timer"
                  type="mast_mdl:Timer_Ref"
                  minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>

   <xs:attribute name="Name"
                 type="mast_mdl:Identifier"
                 use="required"/>
   <xs:attribute name="Speed_Factor"
                 type="mast_mdl:Float"
                 use="optional"/>
   <xs:attribute name="Worst_ISR_Switch"
                 type="mast_mdl:Normalized_Exec_Time"
                 use="optional"/>
   ...
   ...
   <xs:attribute name="Max_Interrupt_Priority"
                 type="mast_mdl:Interrupt_Priority"
                 use="optional"/>
   ...
   <xs:attribute name="Synchronization_Source"
                 type="mast_mdl:Identifier_Ref"
                 use="optional"/>
   <xs:attribute name="System_Timer"
                 type="mast_mdl:Identifier_Ref"
                 use="optional"/>
</xs:complexType>
```

```
<xs:complexType name="Timer_Ref">
   <xs:attribute name="Name"
                 type="mast_mdl:Identifier_Ref"/>
</xs:complexType>
```

Código 3.71 – Tipo complejo `Regular_Processor`

Ejemplo 2. La correspondencia entre la clase `Fork` (Figura 3.75) y el tipo complejo de igual nombre (Código 3.72) ejemplifica las estrategias 0, 5a y 6a.

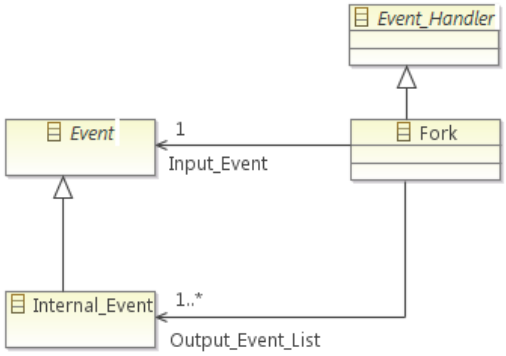


Figura 3.75 – Clase `Fork`

```
<xs:complexType name="Fork">
   <xs:attribute name="Input_Event"
                 type="mast_mdl:Identifier_Ref"
                 use="required"/>
   <xs:attribute name="Output_Events_List"
                 type="mast_mdl:Identifier_Ref_List"
                 use="required"/>
</xs:complexType>
```

Código 3.72 – Tipo complejo `Fork`

Ejemplo 3. La correspondencia entre la clase `Thread` (Figura 3.76) y el tipo complejo de igual nombre (Código 3.73) ejemplifica las estrategias 0, 1, 3a, 3b y 5a.

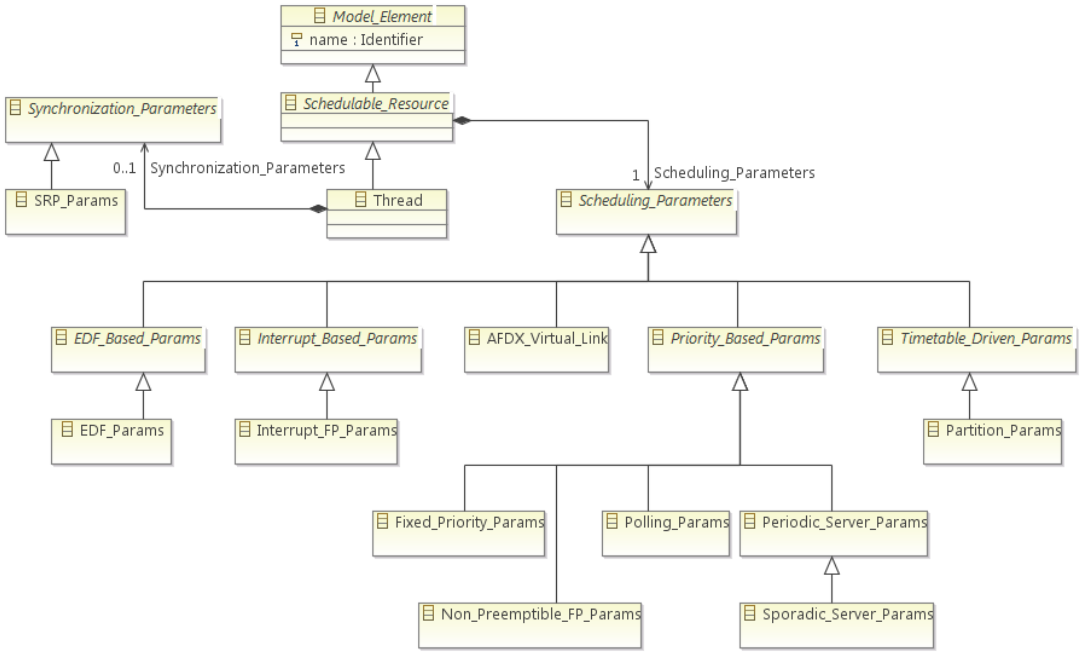


Figura 3.76 – Clase `Thread`

```
<xs:complexType name="Thread">
   <xs:sequence>
      <xs:choice>
         <xs:element name="Interrupt_FP_Params"
                     type="mast_mdl:Interrupt_FP_Params"/>
         <xs:element name="Fixed_Priority_Params"
                     type="mast_mdl:Fixed_Priority_Params"/>
         <xs:element name="Non_Preemptible_FP_Params"
                     type="mast_mdl:Non_Preemptible_"/>
         <xs:element name="Polling_Params"
                     type="mast_mdl:Polling_Params"/>
         <xs:element name="Periodic_Server_Params"
                     type="mast_mdl:Periodic_Server_Params"/>
         <xs:element name="Sporadic_Server_Params"
                     type="mast_mdl:Sporadic_Server_Params"/>
         <xs:element name="EDF_Params"
                     type="mast_mdl:EDF_Params"/>
         <xs:element name="Timetable_Driven_Params"
                     type="mast_mdl:Partition_Params"/>
         <xs:element name="AFDX_Virtual_Link"
                     type="mast_mdl:AFDX_Virtual_Link"/>
      </xs:choice>
      <xs:element name="SRP_Params"
                  type="mast_mdl:SRP_Params"
                  minOccurs="0"/>
   </xs:sequence>
   <xs:attribute name="Name"
                 type="mast_mdl:Identifier"
                 use="required"/>
   <xs:attribute name="Scheduler"
                 type="mast_mdl:Identifier_Ref"
                 use="required"/>
</xs:complexType>
```

Código 3.73 – Tipo complejo `Thread`

3.4.4 Automatización en MAST-2

La recién expuesta implementación en MAST-2 de la estrategia de interoperabilidad se obtiene automáticamente siguiendo el camino presentado en la subsección 3.4.2 previa. Esto significa que primero se ha obtenido el modelo XSD correspondiente a `Mast2_Model.xsd` y a continuación se ha construido el modelo de *mappings* (modelo instructor) que relaciona tal modelo XSD con el metamodelo `Mast2.ecore`. El contenido del modelo de *mappings* refleja las estrategias de formalización empleadas en `Mast2_Model.xsd`. Por último, el desarrollo de las HOT mostradas en la Figura 3.61 permite la generación automática de la herramienta específica de interoperabilidad.

Los apartados a continuación abordan tales modelos.

Modelo XSD correspondiente a Mast2_Model.xsd

Se ofrece a continuación una visión gráfica, a modo de diagramas de objetos, de ciertas partes ilustrativas del modelo XSD obtenido a partir del W3C-Schema `Mast2_Model.xsd`.

En primer lugar, la Figura 3.77 muestra el contenedor principal del modelo, instancia de la clase `XSDSchema`, que contiene cada una de las instancias de las clases `XSDSimpleTypeDefinition` y `XSDComplexTypeDefinition` correspondientes respectivamente a cada tipo simple y complejo declarado en el W3C-Schema.

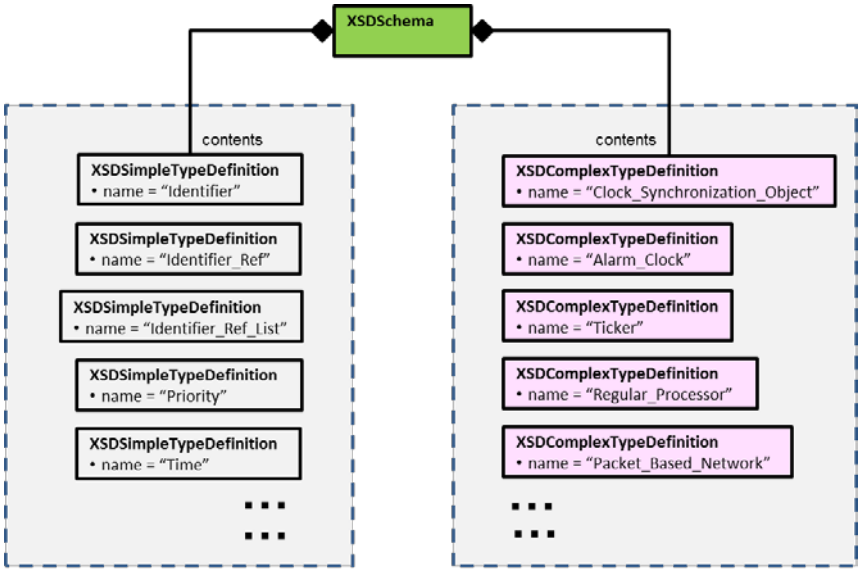


Figura 3.77 – Raíz del modelo XSD correspondiente a `Mast2_Model.xsd`

A continuación, la Figura 3.78, la Figura 3.79 y la Figura 3.80 muestran las partes del modelo correspondientes a los tipos complejos `Regular_Processor`, `Fork` y `Thread` (Código 3.71, Código 3.72 y Código 3.73 respectivamente).

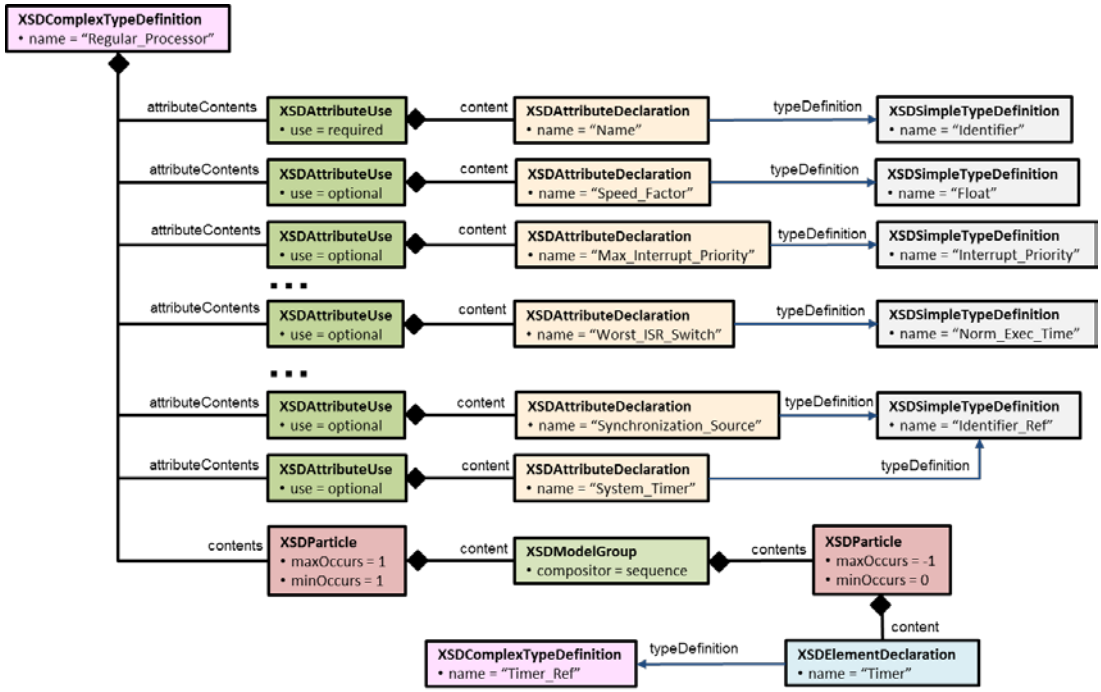


Figura 3.78 – Modelado del tipo complejo `Regular_Processor`

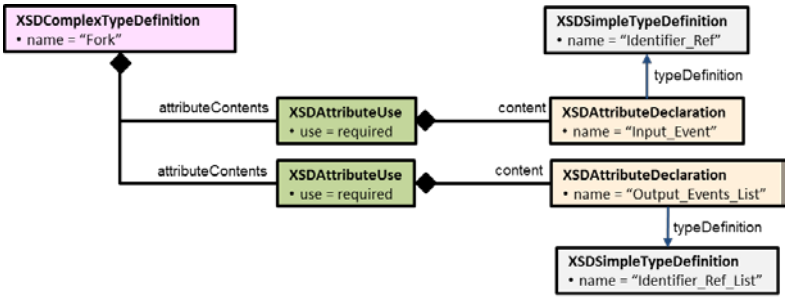


Figura 3.79 – Modelado del tipo complejo `Fork`

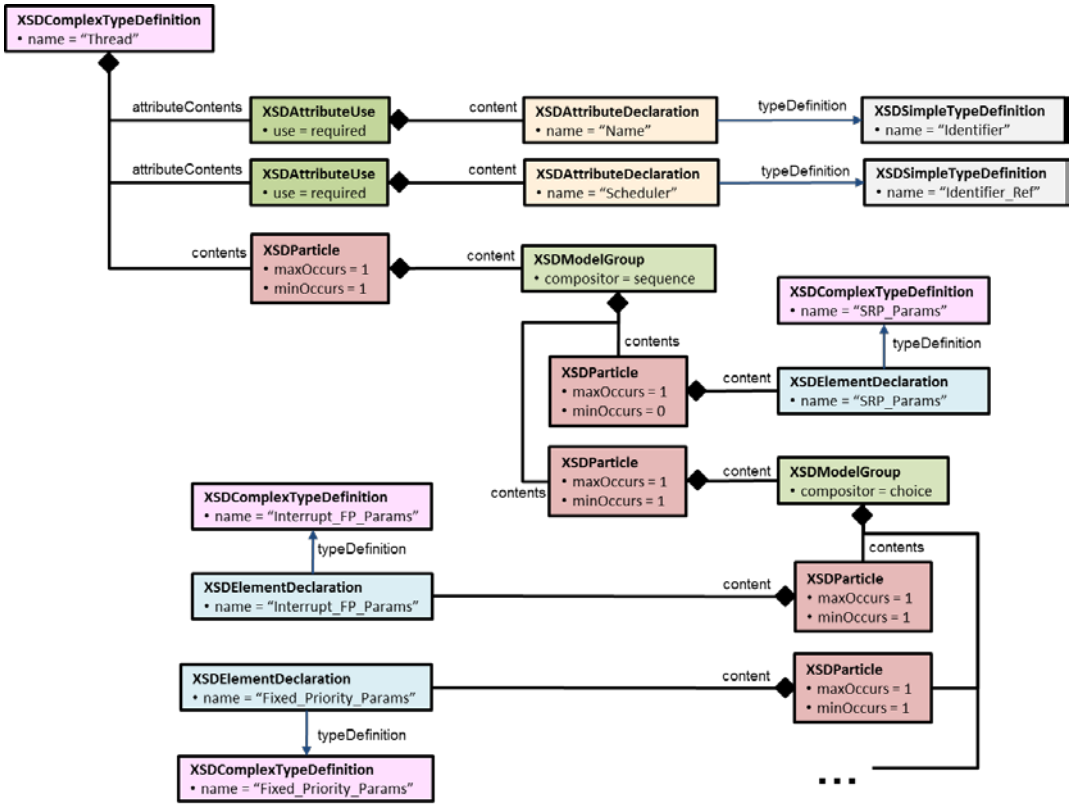


Figura 3.80 – Modelado del tipo complejo `Thread`

EMF adapta especialmente la funcionalidad del *Sample Reflective Model Editor* cuando es empleado para gestionar tales modelos conformes a XSD.`ecore`. Por ejemplo, la Figura 3.81 y la Figura 3.82 muestran abierto en dicho editor el modelo obtenido a partir del W3C-Schema `Mast2_Model.xsd`.

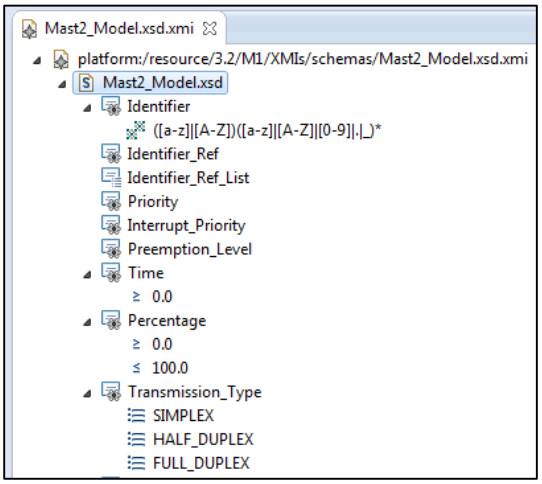


Figura 3.81 – Modelo XSD correspondiente a `Mast2_Model.xsd` (detalle de los tipos simples)

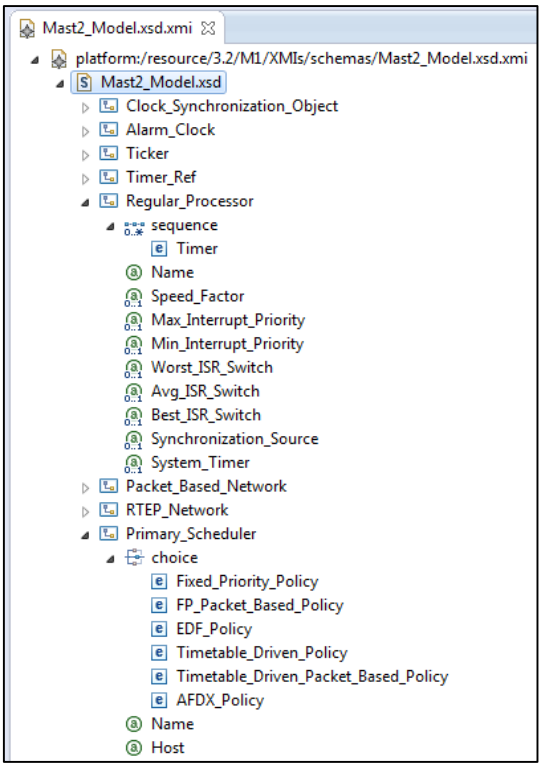


Figura 3.82 – Modelo XSD correspondiente a `Mast2_Model.xsd` (detalle de los tipos complejos)

Modelo de mappings Mast2.ecore ↔ Mast2_Model.xsd

Se ofrece a continuación una visión gráfica, a modo de diagramas de objetos, de ciertas partes ilustrativas del modelo de *mappings* construido para formular las correspondencias entre el metamodelo `Mast2.ecore` y el modelo XSD del apartado anterior.

En primer lugar, la Figura 3.83 muestra el contenedor principal del modelo, instancia de la clase `MappingRoot`, que contiene cada una de las instancias de la clase `Mapping` encargadas de representar las correspondencias entre clases del metamodelo y tipos complejos del W3C-Schema.

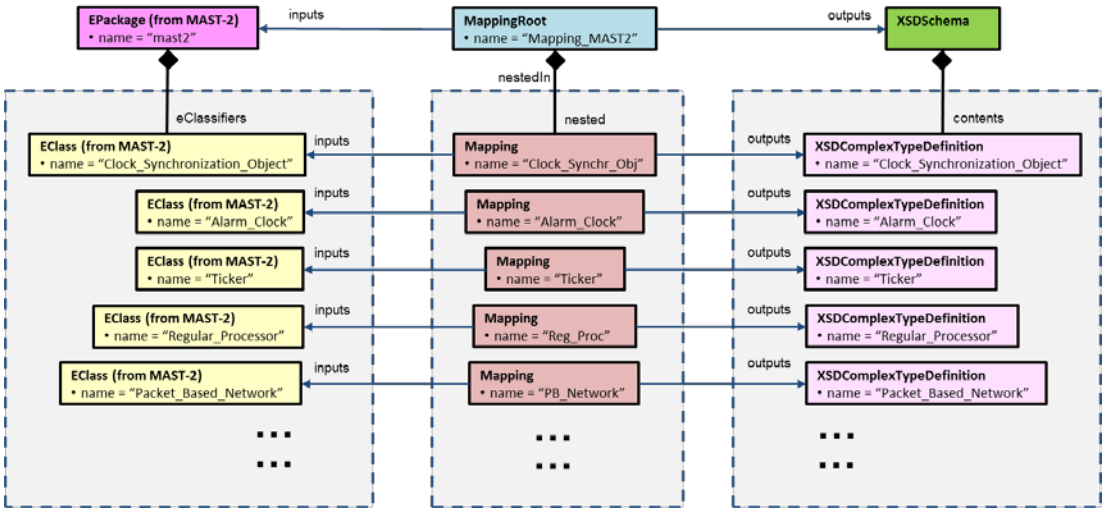


Figura 3.83 – Raíz del modelo de *mappings*

A continuación, la Figura 3.84, la Figura 3.85 y la Figura 3.86 muestran las partes del modelo correspondientes a las correspondencias entre las clases `Regular_Processor`, `Fork` y `Thread` y los tipos complejos de igual nombre.

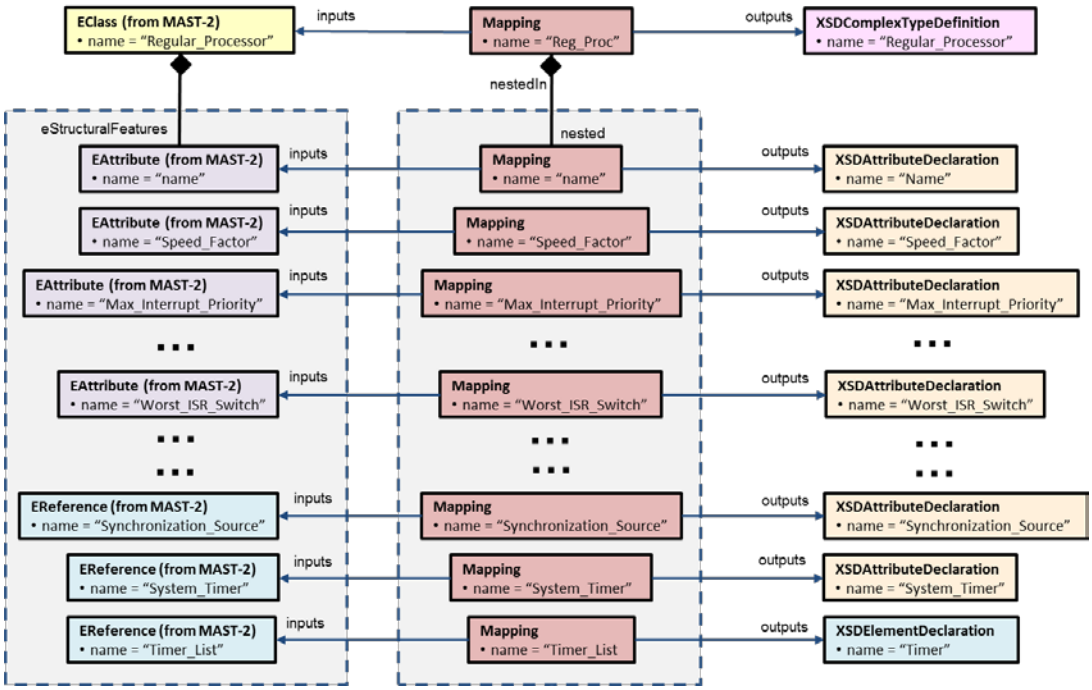


Figura 3.84 – *Mapping* Regular_Processor

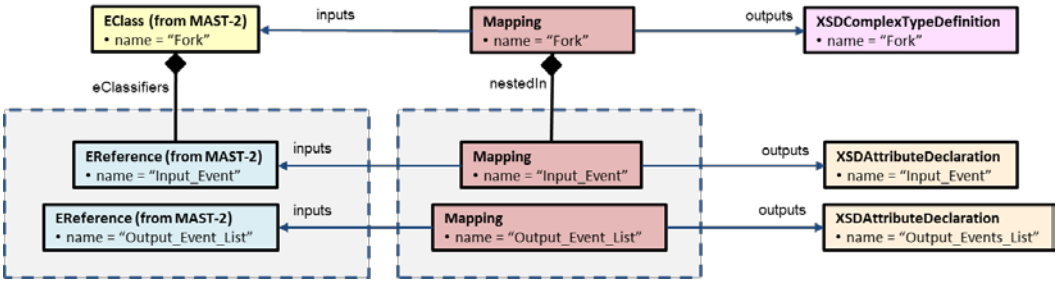


Figura 3.85 – *Mapping* Fork

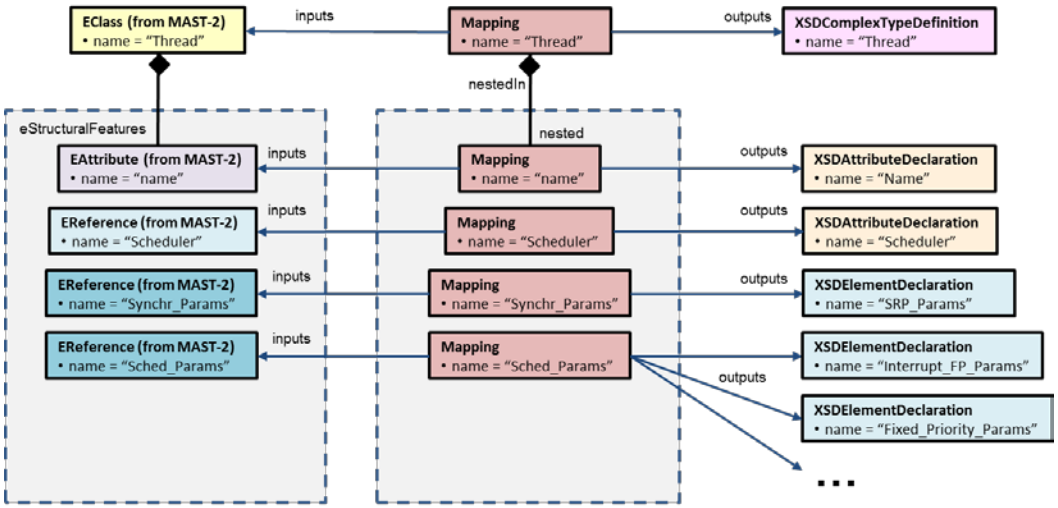


Figura 3.86 – *Mapping* Thread

Al igual que para los modelos conformes a `XSD.ecore`, en este caso EMF también adapta específicamente la funcionalidad del *Sample Reflective Model Editor* cuando es invocado para abrir modelos conformes a `Mapping.ecore`. Por ejemplo, la Figura 3.87 muestra abierto en dicho editor el modelo del ejemplo.

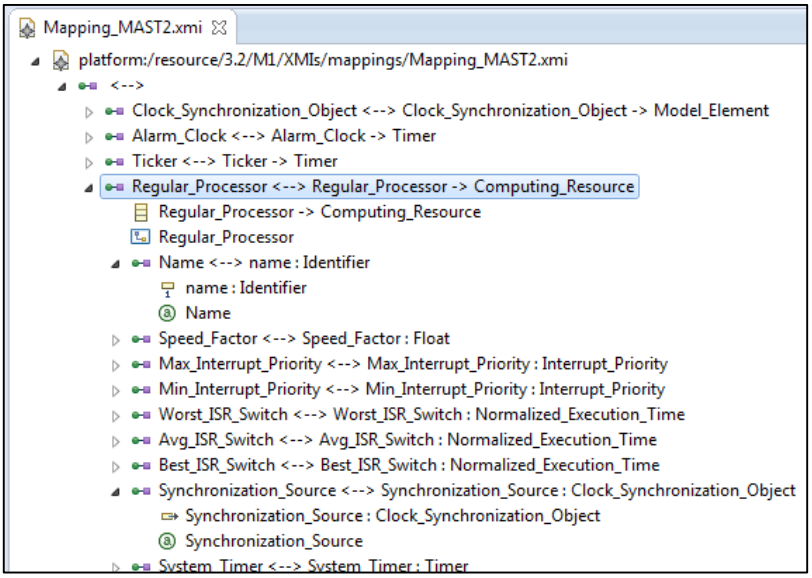


Figura 3.87 – Modelo de *mappings* relacionando Mast2_Model.xsd y Mast2.ecore

3.5 Material desarrollado

Se dedica esta última sección del capítulo a presentar de una manera concisa el material desarrollado en relación a las tres metaherramientas presentadas en las secciones anteriores.

3.5.1 Verificación de cumplimiento de restricciones

En el enlace www.istr.unican.es/members/cesarcuevas/phd/constraintsVerification.html puede encontrarse todo el material desarrollado en relación a la herramienta genérica para verificación de cumplimiento de restricciones, tanto artefactos genéricos como aplicaciones a MAST-2. La Figura 3.88 muestra una visión esquemática de tales artefactos, los cuales ya han sido mencionados puntualmente a lo largo del capítulo.

- **Artefactos propios de la estrategia**:

  1. **CC.ecore**. Formulación Ecore del metamodelo para formalización de los modelos mediante los cuales se formula la información de caracterización de las restricciones especificadas sobre un metamodelo de dominio.

     Se encuentra también disponible su especificación completa y detallada (**CC.pdf**).

  2. **CC_to_ATL.atl**. Implementación ATL de la HOT de síntesis núcleo de la metaherramienta de verificación de modelos.

     Se encuentra también disponible su especificación completa y detallada (**CC_to_ATL.pdf**).

  3. **CVD.ecore**. Formulación Ecore del metamodelo para formalización de los modelos mediante los cuales formular el resultado de verificar otros modelos.

     Se encuentra también disponible su especificación completa y detallada (**CVD.pdf**).

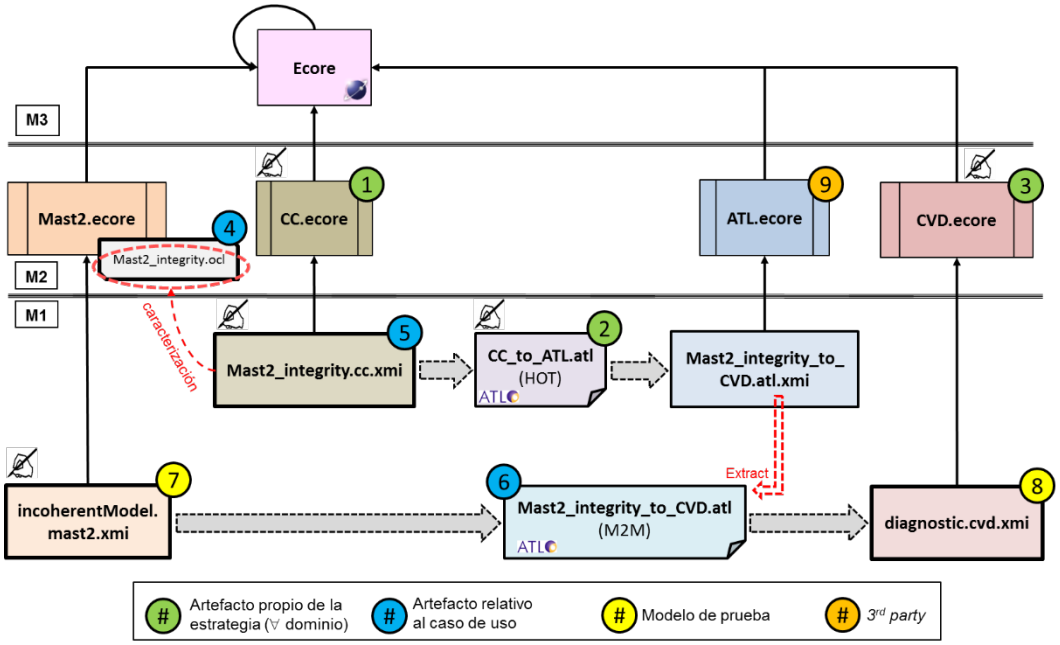


Figura 3.88 – Artefactos correspondientes a la herramienta de verificación

- **Artefactos relativos al caso de uso MAST-2** (metamodelo MAST-2 y sus restricciones de integridad):

  4. **Mast2_integrity.ocl**. Formulación OCL de las restricciones intrínsecas del metamodelo MAST-2 (*Restricciones MAST-2*).
  5. **Mast2_integrity.cc.xmi**. Modelo caracterizador de las *Restricciones MAST-2*.
  6. **Mast2_integrity_to_CVD.atl**. Implementación ATL de la transformación de chequeo correspondiente a las *Restricciones MAST-2*.

- **Modelos de prueba**:

  7. **incoherentModel.mast2.xmi**. Un modelo MAST-2 simple que incurre en la violación de algunas de las *Restricciones MAST-2*.
  8. **diagnostic.cvd.xmi**. Modelo obtenido al aplicar la transformación de chequeo correspondiente a las *Restricciones MAST-2* sobre el modelo anterior.

- **Artefactos *3rd party***.

  9. **ATL.ecore**. Formulación Ecore del metamodelo que formaliza la sintaxis abstracta de ATL.

3.5.2 Construcción de modelos acordes a vistas restrictivas

En el enlace www.istr.unican.es/members/cesarcuevas/phd/constrainingViews.html puede encontrarse todo el material desarrollado en relación a la herramienta genérica para construcción de modelos acordes a vista restrictivas, tanto artefactos genéricos como aplicaciones a MAST-2. La Figura 3.89 y la Figura 3.90 muestran una visión esquemática de tales artefactos, los cuales ya han sido mencionados puntualmente a lo largo del capítulo.

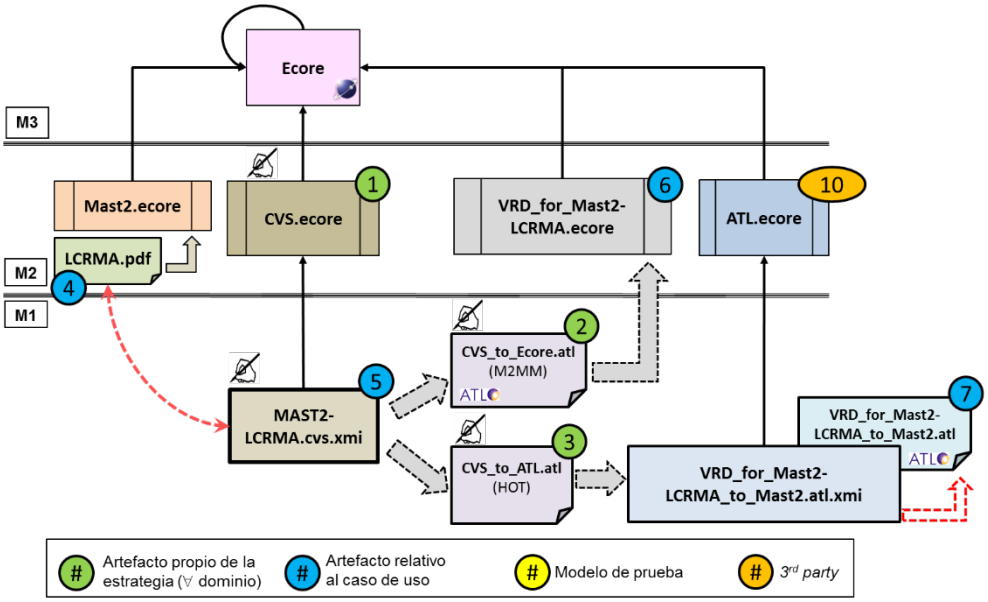


Figura 3.89 – Artefactos correspondientes a la herramienta de construcción (I)

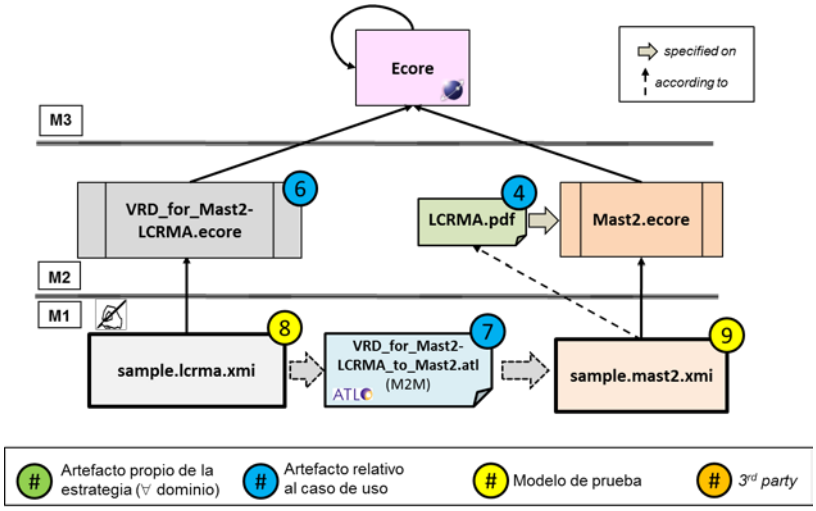


Figura 3.90 – Artefactos correspondientes a la herramienta de construcción (II)

- **Artefactos propios de la estrategia**:

  1. **CVS.ecore**. Formulación Ecore del metamodelo para formalización en forma de modelos de las vistas restrictivas.

     Se encuentra también disponible su especificación completa y detallada (**CVS.pdf**)

  2. **CVS_to_Ecore.atl**. Implementación ATL de la transformación promocionadora que genera cada metamodelo VRD.

  3. **CVS_to_ATL.atl**. Implementación ATL de la HOT que genera cada transformación *VRD_To_Dominio*.

- **Artefactos relativos al caso de uso MAST-2**:

  4. **LCRMA.pdf**. Especificación detallada de la vista LinuxClassicRMA. Se trata de un ejemplo de vista restrictiva especificada sobre un metamodelo de dominio, en este caso MAST-2. Como para cualquier vista, toma como base el metamodelo cuya estructura se desea

restringir (MAST-2) y explicita aquellas clases de las cuales se permiten instancias. La especificación también define las categorías mediante las que se formalizan las condiciones restrictivas que se imponen sobre las instancias de dichas clases permitidas (valores de atributos, referencias a *null*, etc.) y estipula los elementos que obligatoriamente han de existir en todo modelo acorde a la vista.

5. **LCRMA.cvs.xmi**. Formulación como modelo CVS de la vista LCRMA.
6. **VRD_for_Mast2-LCRMA.ecore**. Formulación Ecore del metamodelo VRD correspondiente a la vista LCRMA.
7. **VRD_for_Mast2-LCRMA_to_Mast2.atl**. Implementación ATL de la transformación que convierte los modelos conformes al metamodelo anterior a modelos MAST-2.

- **Modelos de prueba**:

8. **sample.lcrma.xmi**. Ejemplo de modelo conforme al metamodelo VRD correspondiente a la vista LCRMA.
9. **sample.mast2.xmi**. Modelo MAST-2 acorde a LCRMA obtenido a partir del modelo anterior.

- **Artefactos *3rd party***.

10. **ATL.ecore**.

3.5.3 Interoperabilidad XML ↔ Modelware

En el enlace www.istr.unican.es/members/cesarcuevas/phd/XML-Modelware.html puede encontrarse todo el material desarrollado en relación a la herramienta genérica para interoperabilidad XML ↔ Modelware, tanto artefactos genéricos como aplicaciones a MAST-2. La Figura 3.91 y la Figura 3.92 muestran una visión esquemática de tales artefactos, los cuales ya han sido mencionados puntualmente a lo largo del capítulo.

- **Artefactos propios de la estrategia**:

1. **Mapping_to_ATL_for_XML_to_Modelware.atl**. Implementación ATL de la HOT que genera las transformaciones de XML a Modelware.
2. **Mapping_to_ATL_for_Modelware_to_XML.atl**. Implementación ATL de la HOT que genera las transformaciones de Modelware a XML.

- **Artefactos relativos al caso de uso MAST-2**:

3. **Mast2_Model.xsd.xmi**. Formulación como modelo XSD del W3C-Schema `Mast2_Model.xsd`.
4. **Mast2.mapping.xmi**. Formulación como modelo conforme al metamodelo `Mapping.ecore` de las correspondencias entre `Mast2_Model.xsd` y `Mast2.ecore`.
5. **XML_to_Modelware_for_Mast2.atl**. Implementación ATL de la transformación que genera modelos conformes a `Mast2.ecore` a partir de modelos conformes a `XML.ecore` (inyectados desde documentos XML conformes a `Mast2_Model.xsd`).

6. **Modelware_to_XML_for_Mast2.atl**. Implementación ATL de la transformación que genera modelos conformes a `XML.ecore` a partir de modelos conformes a `Mast2.ecore`.

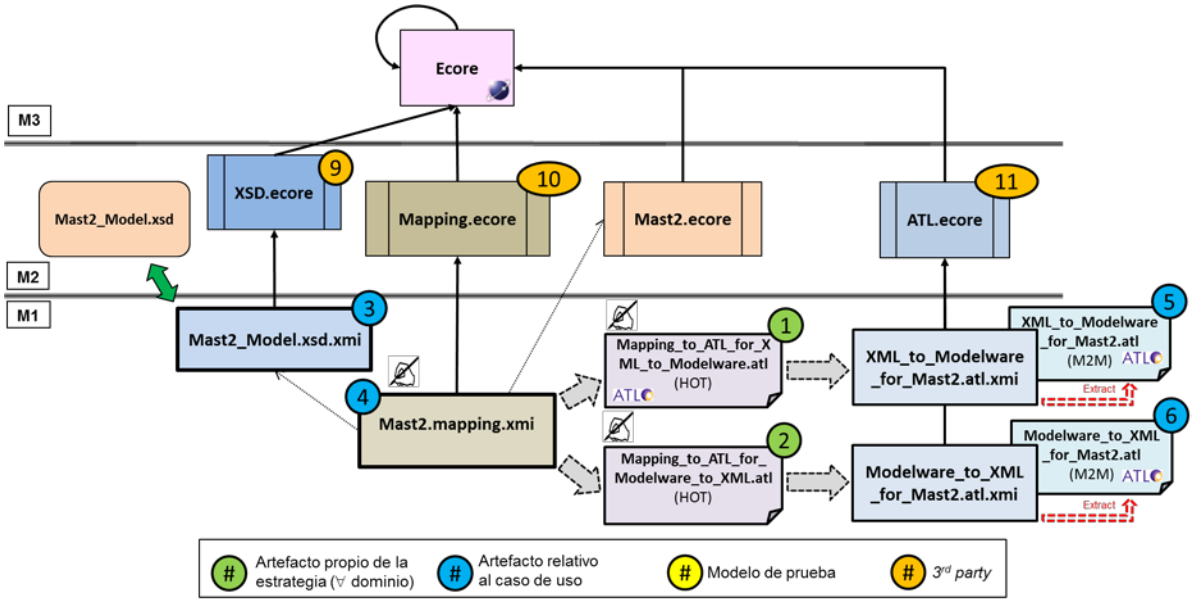


Figura 3.91 – Artefactos correspondientes a la herramienta de interoperabilidad (I)

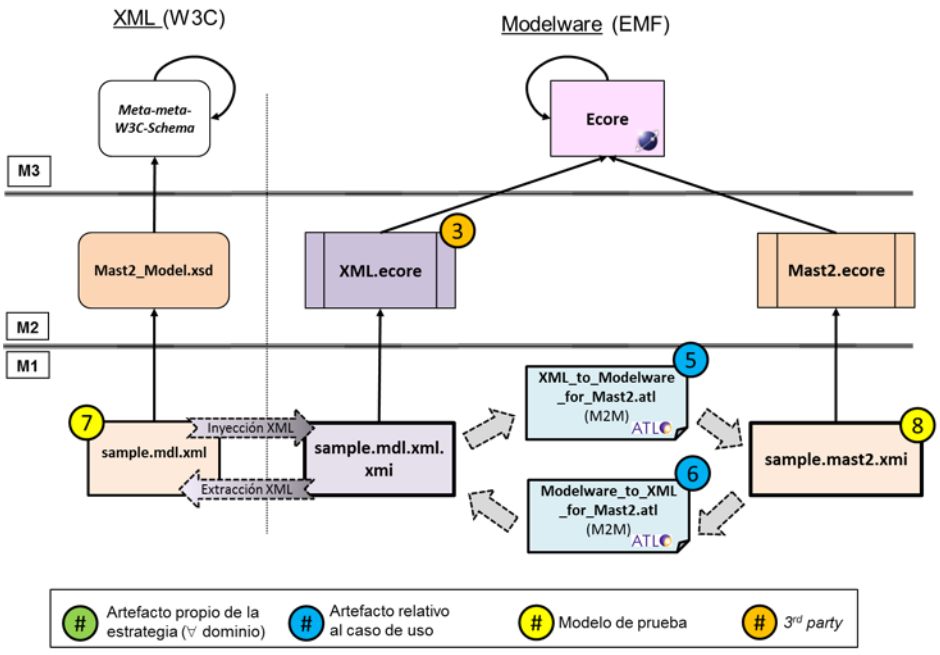


Figura 3.92 – Artefactos correspondientes a la herramienta de interoperabilidad (II)

- **Modelos de prueba**:

  7. **sample.mdl.xml**. Ejemplo de modelo MAST-2 expresado como documento XML conforme a `Mast2_Model.xsd`.
  8. **sample.mast2.xmi**. Ejemplo de modelo conforme a `Mast2.ecore`.

- **Artefactos *3rd party***.

9. **XSD.ecore**. Formulación Ecore proporcionada por EMF del metamodelo que formaliza la especificación W3C-Schema.
10. **Mapping.ecore**. Formulación Ecore proporcionada por EMF del metamodelo empleado para formalizar modelos de correspondencias o *mappings*.
11. **ATL.ecore**.

4 Modelo de referencia para la implementación de entornos MDSE de desarrollo de aplicaciones software

En el capítulo 1 se planteó como objetivo central de la Tesis facilitar la adopción de la MDSE a los expertos en Ingeniería Software que diseñan entornos y, de forma más específica, proporcionar la capacidad de adaptar, extender y evolucionar los entornos en base a la formulación de modelos específicos del dominio, y no al desarrollo de código basado en el conocimiento y uso de la infraestructura MDSE que da soporte al entorno.

Con este fin, en el capítulo 2 se han propuesto estrategias para la formulación de la información y en el capítulo 3 estrategias para el desarrollo de herramientas que pueden adaptarse y mantenerse formulando o modificando los modelos que adaptan su funcionalidad a nuevas situaciones.

En este capítulo se extiende el objetivo a la formulación, diseño e implementación del propio entorno, de los procesos que le proporcionan su funcionalidad y de las opciones de interacción, supervisión y control que se le asignan.

Un entorno tiene que proporcionar tres funcionalidades básicas para el desarrollo de proyectos software:

- **Servir de repositorio** (persistente o temporal) **de los modelos** que contienen la información introducida, producida o resultante relativa al proyecto software bajo desarrollo.
- **Dar soporte a las herramientas** con las que se introduce, supervisa, transforma y almacena la información en los diferentes procesos de desarrollo de sistemas software.
- **Proporcionar recursos de interacción** para que el desarrollador pueda supervisar, dirigir y controlar la ejecución secuencial o iterativa de los procesos de desarrollo.

En este capítulo se aborda la definición de un modelo de referencia genérico para diseñar entornos de desarrollo, así como la de un conjunto de recursos de soporte que faciliten al experto diseñador de entornos su especificación e implementación.

El modelo de referencia que se propone ha sido desarrollado identificando aquellas capacidades básicas, tanto funcionales como de interacción, que son comunes a cualquier entorno y diseñando un metamodelo que formaliza la información que han de contener los modelos que describan el entorno.

Así, de acuerdo con el espíritu de la disciplina MDSE, la especificación y diseño de un entorno va a consistir fundamentalmente en la elaboración de modelos, y no en el desarrollo de su código de implementación.

Se considera que el ingeniero que utiliza un entorno para desarrollar proyectos software organiza su actividad mediante procesos que consisten en la ejecución secuencial o iterativa de operaciones más básicas (tareas) y ejecutados bajo su supervisión y control.

En el modelo de referencia propuesto, los procesos se formulan mediante modelos que describen su secuencia de tareas constituyentes y especifican la naturaleza de éstas. A su vez, las tareas se formulan mediante nuevos modelos que describen su naturaleza, los modelos sobre los que operan, las transformaciones involucradas, la información que se ofrece al

operador para la toma de decisiones y las opciones de control con las que éste supervisa, valida y dirige su ejecución.

En este capítulo se define el modelo de referencia que incluye una concepción de entorno y un metamodelo que formaliza los modelos para describir los procesos y las tareas. Asimismo, como prueba de concepto se ha diseñado e implementado una herramienta que los interpreta, y en base a ellos, genera automáticamente los recursos del entorno diseñado. Esta concepción de entornos bajo el paradigma dirigido por modelos, será referida en lo sucesivo como ***MDDE*** (*Model-Driven Development Environment*).

Los objetivos que guían el diseño de cualquier entorno acorde a *MDDE* son:

- **Integrar de forma natural los múltiples aspectos y dominios que conciernen al diseño de un sistema *software*** (especificación funcional y no funcional, diseño arquitectural o detallado, codificación, despliegue sobre la plataforma de ejecución, configuración funcional, no funcional o de despliegue, etc.).
- **Hacer sencillo su uso a los diseñadores de sistemas software**, ofreciendo una estrategia de acceso a los diferentes dominios (vistas), información (modelos) y procesos (herramientas) estandarizada e integrada.
- **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 interoperatividad** del entorno.

MDDE se describe a través de un modelo de referencia o *framework*, el cual representa una abstracción que especifica sus elementos constituyentes, tanto conceptuales (caracterizados por el papel que juegan) como funcionales (caracterizados por los servicios que prestan y la interfaz que ofrecen), define la arquitectura con la que se integran y los mecanismos de interacción entre ellos.

Este *framework* constituye un mapa conceptual independiente de cualquier implementación y proporciona al diseñador de sistemas software la información necesaria para trabajar con un entorno *MDDE* y al diseñador de entornos la vista arquitectural que dirige los elementos que diseña. Aunque se formule de manera agnóstica respecto a cualquier plataforma, para su validación requiere ser implementado sobre una plataforma concreta. Se ha elegido la plataforma Eclipse porque:

- Su *framework* de modelado EMF/Ecore soporta de forma natural la disciplina MDSE.
- La infraestructura del *workbench* de Eclipse posibilita una implementación inmediata y estandarizada.
- Por su arquitectura modular (basada en *plugins)* hace que el mecanismo de integración de nuevas funcionalidades sea robusto y confiable.

Así, la implementación de *MDDE* sobre Eclipse se denomina ***MDDE-Eclipse***, lo cual es ortogonal al dominio y/o metodología a que se pretenda dar soporte. Por ejemplo, podría hablarse de una implementación de *MDDE* sobre la plataforma Eclipse y concebida para dar soporte al desarrollo de SDTREs basado en MAST, con lo cual se trataría del entorno ***MDDE-Eclipse-MAST***.

El modelo de referencia de *MDDE* se expone desde tres puntos de vista. La Figura 4.1 esquematiza tales vistas parciales.

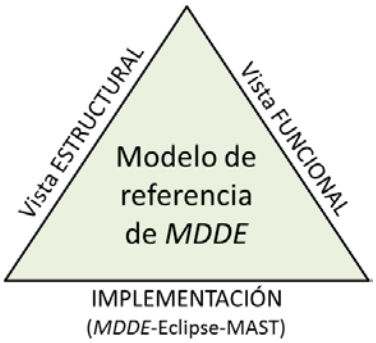


Figura 4.1 – Diferentes vistas del modelo de referencia *MDDE*

En el resto del capítulo se desarrollan los tres puntos de vista:

- La sección 4.1 define, desde un **punto de vista estructural,** los elementos conceptuales que constituyen el *framework*, las relaciones entre ellos y su formalización como metamodelo.
- La sección 4.2 aborda el modelo de referencia desde un **punto de vista funcional**, definiendo los elementos que constituyen el motor interno del entorno y las interfaces que utiliza.
- La sección 4.3 analiza el modelo de referencia desde el **punto de vista de la implementación**, describiendo las líneas generales y los detalles más significativos de la realización *MDDE-Eclipse-MAST*.

4.1 Modelo de referencia: elementos conceptuales y estructura

En esta sección se describe el modelo de referencia de *MDDE* desde un punto de vista estructural y a dos niveles:

- **Descripción de los elementos conceptuales**, en base a sus atributos e interrelaciones, especificando para cada uno de ellos sus tipos derivados y su clasificación (por la naturaleza de su especialización, por su persistencia o por su función específica).
- **Formulación del metamodelo** que formaliza los aspectos estructurales del entorno conceptual y sirve de base para la descripción de cada entorno concreto.

4.1.1 Elementos conceptuales del *framework* de MDDE

La Figura 4.2 identifica los elementos conceptuales del modelo de referencia de *MDDE* que se describen en esta sección.

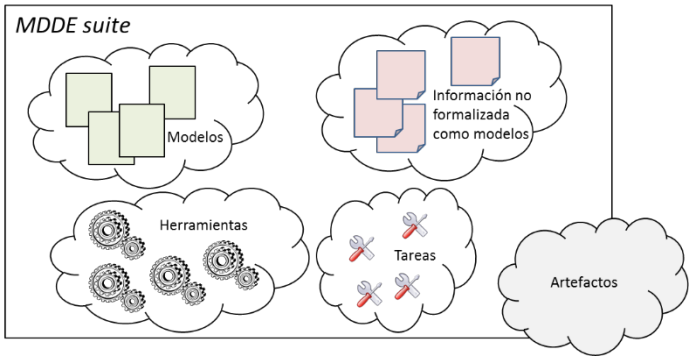


Figura 4.2 – Un entorno *MDDE* y sus elementos conceptuales

MDDE suite

Es el elemento raíz (contenedor) del entorno, el ente que se distribuye, se instala, sirve de espacio de trabajo y proporciona persistencia a los elementos que se crean cuando se trabaja en él. Básicamente, se compone de tres tipos de elementos:

- **Modelos**. Constituyen los elementos contenedores de cualquier tipo de información presente en un entorno *MDDE*, ya sea información que describe el propio entorno como la información que describe al sistema bajo desarrollo (SUD).
- **Interfaz gráfica de interacción con el operador**. Ofrece un conjunto de marcos (textuales, tabulares y gráficos) a través de los que el operador lee y accede a la información contenida en un entorno *MDDE*, así como también controles intuitivos a través de los que gestiona el entorno mediante una estrategia reactiva.
- **Gestores de modelos**. Constituyen el soporte funcional de interpretación de los modelos de acuerdo a su naturaleza. Representan aplicaciones embebidas en el entorno, que le proporcionan la operatividad que se describe en el modelo de referencia y que se invocan en el entorno en base a los eventos que se generan sobre él y el contexto en el que ocurren. Se describen en la sección 4.2, relativa a los elementos funcionales del entorno.

La concepción *MDDE* es única, pero el contenido nativo con que se distribuye un entorno *MDDE* (herramientas y modelos/metamodelos, así como elementos funcionales) determina su capacidad de evolución y enriquecimiento futuro. En base a ello, se distinguen los siguientes tipos de entornos *MDDE*:

- **Entorno *MDDE* especializado**. Entorno sencillo creado para dar soporte a uno o más dominios específicos en el desarrollo de sistemas *software* (por ejemplo, entorno para el análisis de planificabilidad de SDTREs). Tal conjunto de dominios se establece en la configuración del entorno y limita su ámbito de aplicación. Un entorno *MDDE* especializado está dotado del contenido nativo para dar soporte a la realización de procesos previstos en su dominio, pero no tiene capacidad de crear nuevos metamodelos o herramientas.
- **Entorno *MDDE* de propósito general**. Entorno dotado de contenido nativo que proporciona soporte al desarrollo de sistemas *software* en un determinado conjunto de dominios y que también ofrece los recursos necesarios para que expertos en nuevos dominios puedan crear en él nuevos metamodelos y herramientas que den soporte a nuevos procesos de desarrollo, extendiendo así persistentemente la funcionalidad del entorno a los nuevos dominios.
- **Entorno *MDDE* constructor**. Entorno dotado del contenido nativo que permite crear nuevos entornos distribuibles, con los recursos, elementos nativos y elementos funcionales que se hayan considerado apropiados para sus usuarios.

Modelos

Toda la información manejada (creada, procesada, analizada, supervisada, etc.) en un entorno *MDDE* está formulada como modelos[59] conformes a metamodelos existentes en él. La función de tales modelos es diversa, clasificándose en:

[59] El mecanismo de persistencia de los modelos no es relevante en el modelo de referencia, pero debe ser único, estar bien definido y corresponder a un formato estándar (por ejemplo XMI) para que los modelos puedan ser intercambiados con otros entornos, sean o no *MDDE*.

- **Modelo de dominio**. Contiene información relativa a un aspecto o punto de vista de un SUS.
- **Modelo de herramienta**. Describe la operatividad, la configuración o el estatus de una herramienta definida en el entorno.
- **Modelo de interacción**. Sirve para transferir una información al operador o aceptar la información que éste introduce.
- **Metamodelo**. Su función es describir la información que pueden contener otros modelos; modelos conformes a él. En *MDDE* se considera que el metamodelo de un metamodelo (metametamodelo) es único en el entorno (por ejemplo, Ecore) y es conforme respecto a él mismo.

Por otro lado, ortogonalmente a la clasificación anterior y atendiendo al ciclo de vida de un modelo en el entorno, se distinguen tres tipos de modelos:

- **Modelo nativo.** Es incorporado al entorno durante su creación y por tanto distribuido de inicio con él. Dado su carácter nativo, son modelos no modificables y no eliminables, aunque pueden ser supervisados o copiados por el operador.

  El conjunto de modelos nativos, junto al resto de elementos nativos, constituye la base de la capacidad funcional, evolutiva y de extensibilidad de un entorno *MDDE.*
- **Modelo de proyecto**. Es producido bajo demanda del operador o por requerirlo la estrategia de desarrollo con información que es relevante para el proyecto bajo desarrollo. Puede ser generado bien mediante construcción manual o bien generado mediante la invocación de procesos en el propio entorno, pero en cualquier caso es almacenado persistentemente en el espacio de datos del entorno. El operador o los procesos pueden modificarlos o eliminarlos, pero si esto último no ocurre, el entorno los salva persistentemente cuando se cierra el proyecto y los recupera cuando se abre de nuevo para continuar con el mismo proyecto.
- **Modelo temporal**. Modelo efímero creado transitoriamente para manejar información interna dentro de la actividad de una herramienta o proceso del entorno. Si la funcionalidad de la herramienta lo permite, estos modelos pueden ser supervisados y modificados por el operador en aquellas fases de su ciclo de vida en las que son accesibles para él, pero siempre son destruidos cuando finaliza la herramienta que los genera.

Información no formalizada como modelos

El ámbito externo a un entorno *MDDE* es heterogéneo, y por ello, no se puede, ni conviene mantener que la información ha de estar formulada como modelos conformes a metamodelos conocidos desde el entorno. Por ello, en la concepción *MDDE* se ha considerado conveniente dar soporte a la formulación de información mediante lenguajes textuales, y así posibilitar:

- **El intercambio de información** con otros entornos o con sistemas que tienen su propio lenguaje específico.
- **Un mecanismo sencillo para que colaboradores humanos sin acceso al entorno le aporten información o la reciban de él**.
- **Proporcionar un medio perdurable a la información** si se prevé que su duración en el tiempo va a ser más prolongada que la supervivencia del propio entorno.
- **Generar modelos sencillos de configuración de sistemas o aplicaciones**.

Los lenguajes textuales formales son el medio habitual de intercambio de información entre los operadores humanos y las aplicaciones informáticas. El lenguaje debe tener una sintaxis abstracta formal y un vocabulario preciso e inequívoco, para que pueda ser gestionado por el computador, a través de analizadores, secuenciadores y editores. Por otro lado, para que el lenguaje pueda ser sea atractivo al operador humano, debe expresar los conceptos de una forma muy próxima a como lo hace en su lenguaje natural propio del dominio de que se trate.

En *MDDE* se contempla la importación o exportación de información textual de dos tipos:

- **Lenguajes de tipo XML formalizados a través de plantillas W3C-Schema**. Éstos son actualmente de uso común debido a:
  - Ser compatibles con cualquier plataforma.
  - El gran número de herramientas y librerías disponibles para llevar a cabo su gestión.
  - Ser fácilmente interpretados por personas humanas.
- **Lenguajes específicos generados a partir de Xtext** y que partiendo de una sintaxis abstracta definida en forma de metamodelo, pueden generar de forma muy sencilla herramientas y librerías que den soporte al lenguaje definido dentro de él.

Herramientas

Una herramienta *MDDE* es un proceso u operación de alto nivel ofrecido por un entorno *MDDE* para algún objetivo de desarrollo o de gestión propia, que enriquece, supervisa o procesa la información contenida en él, y que es invocada explícitamente por el operador desde el propio entorno.

Como se muestra en la Figura 4.3, una herramienta se compone de una secuencia lineal y ordenada de *tareas* que son actividades más elementales definidas de forma individual en el entorno para facilitar su reutilización en otras herramientas.

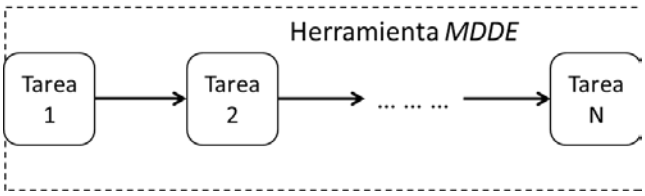


Figura 4.3 – Una herramienta *MDDE* como secuencia de tareas

Las herramientas *MDDE* permiten al operador:

- **Incorporar al entorno, de forma asistida, información relativa al SUD**, introduciendo nuevos modelos que enriquecen la información ya existente.
- **Presentar la información del SUD disponible en el entorno** de acuerdo con el punto de vista que incorpora la herramienta que se utiliza.
- **Procesar la información del SUD disponible en el entorno** de acuerdo con estrategias de transformación, integración o análisis incorporadas por la herramienta que se invoca.
- **Gestionar, de forma asistida, los modelos del entorno**, organizando, almacenando persistentemente o eliminando los elementos dentro de él.

- **Importar y exportar la información del SUD** desde o hacia otros formatos diferentes a los del entorno.

Toda herramienta *MDDE* se formula completamente a través de un modelo, por lo que su incorporación a un entorno *MDDE* se realiza registrando en él su modelo. Algunas herramientas son nativas, puesto que sus modelos se han introducido en el entorno en su creación y permanecen inalteradas durante todo el ciclo de vida de éste, proporcionándole una funcionalidad operativa básica. Por otro lado, el operador (en el rol de desarrollador de dominios) puede definir nuevas herramientas aportando al entorno los correspondientes modelos que las describen o modificando los modelos de otras herramientas ya existentes. La Figura 4.4 muestra la introducción de una nueva herramienta.

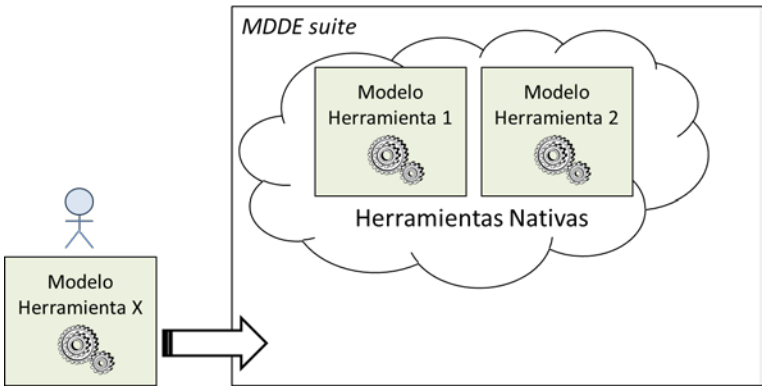


Figura 4.4 – Incorporación de una nueva herramienta a un entorno *MDDE*

El modelo que formula una herramienta *MDDE* describe:

- **Su funcionalidad**, especificando la secuencia de tareas que pueden ser invocadas durante su ejecución.
- **Su contexto de invocación**, es decir, el elemento del entorno desde el que puede ser invocada, bien directamente (menú o botón de herramientas) o bien mediante selección contextual sobre proyecto, carpeta o fichero.
- **Las opciones de configuración** que ofrece al operador que la invoca.

  Los parámetros de configuración de una herramienta están definidos como un modelo que contiene la identificación de los parámetros, su tipo, el valor que se les asigna (siempre tienen definidos al menos valores por defecto) y si pueden ser modificados o no por el operador. El marco de gestión de una herramienta, en base a su modelo, asiste al operador a configurar su ejecución, si lo desea.

- **La información de estado que ha de ir generando** mientras se está ejecutando.

  El estado de ejecución se describe también como un modelo, cuyos valores son establecidos por la herramienta de acuerdo con los resultados que se obtienen en su ejecución, y proporcionan información relevante respecto al éxito o fracaso de la ejecución de las tareas internas y respecto a los modelos transitorios y/o persistentes que va generando la ejecución de la herramienta.

La ejecución de una herramienta *MDDE* se realiza siempre bajo el control y supervisión del operador. Tras la invocación, hay al menos tres puntos en los que se requiere su intervención:

1. **Establecer o aceptar la configuración** de lanzamiento de la herramienta.
2. **Lanzar su ejecución**.
3. **Dar por concluida la ejecución** (lo que supone la eliminación de toda la información de estatus y modelos transitorios que se hayan generado).

Cuando éstos son los únicos pasos de la ejecución, se habla de ejecución en *modo continuo*. El otro modo posible es *paso a paso*, en el cual la ejecución de la herramienta requiere que el operador intervenga en la ejecución de cada tarea integrante, bien para reconfigurar su ejecución específica, para repetir la ejecución de tareas previas o puentear la ejecución de tareas siguientes.

Para posibilitar la ejecución supervisada de una herramienta, cuando se invoca se despliega en el entorno un marco de interacción con los controles necesarios para la gestión de su ejecución. Este marco se puebla en base al modelo que describe a la herramienta, mostrando la secuencia de tareas constituyentes y posibilitando, en colaboración con la ventana de propiedades, la asignación de valores a los parámetros de configuración caracterizados como modificables, tanto aquellos propios de la herramienta como de sus tareas.

Artefactos (gadgets)

Un artefacto es un recurso software utilizado por las herramientas de un entorno *MDDE* pero que ha sido desarrollado fuera de él, con independencia de su modelo de referencia, pudiendo incluso tratarse de material legado. Tanto el empleo de un artefacto por parte de una herramienta como su interacción con los recursos del entorno no se realizan de manera directa, sino que el artefacto precisa de un adaptador que permita su invocación, configuración, manifestación de estatus e interacción con el operador.

Normalmente un artefacto proporciona una funcionalidad de:

a. Procesamiento de modelos.
b. Presentación de la información.
c. Interacción con el operador.

Un artefacto puede ser:

a. **Interno**. Se puede ejecutar en el espacio de memoria del propio entorno *MDDE* en que se invoca.
b. **Externo**. Se tiene que ejecutar en un espacio de memoria distinto al del entorno *MDDE* desde donde se invoca, bien en el mismo nudo de procesamiento que el entorno o en otro procesador diferente (**artefacto externo remoto**).

En el caso de los artefactos externos, su gestión e interacción con el entorno se realizan a través de mecanismos de comunicación establecidos en el modelo de referencia de *MDDE* y proporcionados por la plataforma en que se ejecuta el entorno. Asimismo, para su invocación se requiere que en el espacio de memoria en que se ejecuta haya una aplicación *demonio* de lanzamiento de artefactos (*GadgetLauncher*).

La Figura 4.5 muestra los elementos de gestión de un artefacto externo.

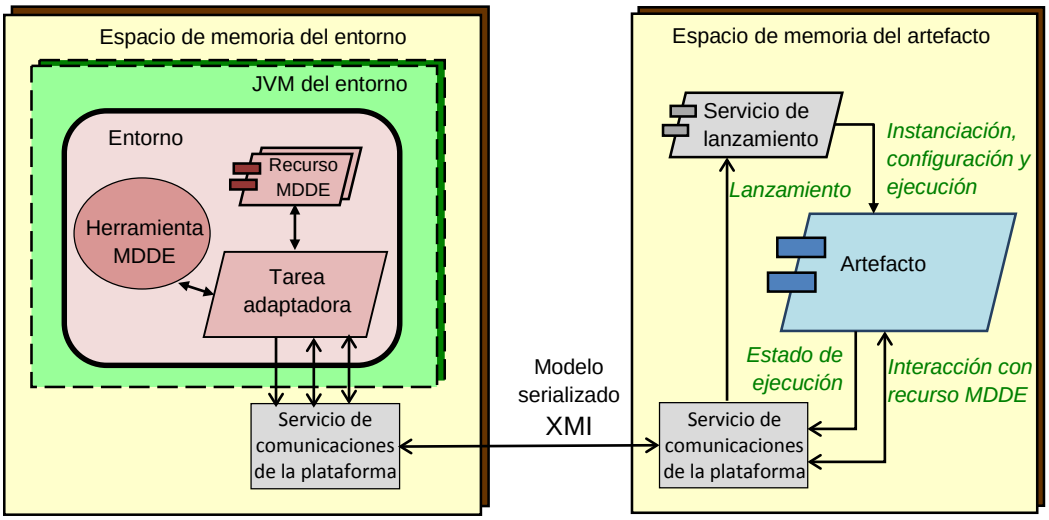


Figura 4.5 – Elementos que participan en la gestión de un artefacto externo

Tareas

En el *framework* de *MDDE*, el elemento operativo básico, representativo de alguna operación útil para el desarrollo de sistemas o para la gestión del propio entorno, se denomina *tarea*. Por tanto, las tareas son las actividades elementales que constituyen las herramientas que operan en el entorno y se introducen con el objetivo de:

- **Incrementar la reusabilidad de funciones** que son compartidas por diferentes herramientas, buscando que tales operaciones se definan una única vez en el entorno. Por ello, la funcionalidad de las tareas es reducida y configurable.
- **Ofrecer una interfaz de gestión homogénea** que facilite la composición y el encadenamiento de múltiples tareas en las herramientas y la interacción con los recursos del entorno.
- **Constituir un adaptador al modelo de referencia *MDDE*** para los elementos desarrollados con independencia de él (artefactos).

Según la forma en que una tarea implementa su funcionalidad, se pueden clasificar como:

- **Tarea nativa.** Es aquella que está definida en el entorno y sólo requiere su invocación directa.
- **Tarea con funcionalidad definida mediante un modelo**. Son tareas nativas que implementan una determinada funcionalidad, pero que requieren de un modelo instructor para adaptar su funcionalidad a los modelos sobre los que ha de operar en la herramienta. Por ejemplo, la tarea puede implementar la funcionalidad definida en una transformación M2M, y el modelo que define su funcionalidad es la descripción de la transformación.
- **Tarea adaptadora de artefacto.** Es una tarea de adaptación entre el entorno y un artefacto, que por su naturaleza, puede ser ejecutado en la plataforma de ejecución del entorno. Proporciona los modelos de entrada que requiere el artefacto en el formato adecuado, almacena en el entorno los modelos de resultados que genera y traduce los mensajes de control y estatus entre el entorno y el artefacto.
- **Tarea adaptadora de artefacto externo**. Es una tarea de adaptación entre el entorno y un artefacto, que por su naturaleza requiere ser ejecutado en una plataforma diferente a la del

entorno. En este caso se requiere la intervención del servicio de comunicación de la plataforma (*GadgetLauncher*) así como la serialización de los modelos y mensajes que se intercambian entre la plataforma y el artefacto.

Por otro lado, en base a su ciclo de ejecución, una tarea *MDDE* puede ser atómica o interactiva:

- **Tarea atómica**. Su ejecución se realiza únicamente en base a los datos de configuración iniciales, por lo que desde el punto de vista del operador se ejecuta atómicamente, sin requerir su intervención. La información sobre los resultados de su ejecución se deduce de la información de estatus que retorna cuando acaba su ejecución.
- **Tarea interactiva**. Su ejecución requiere la intervención del operador, tanto para gestionar (introducir, modificar o eliminar) información como para decidir las opciones de su flujo de control.

  Una tarea interactiva proporciona el mecanismo con el que interacciona el operador, pudiendo ser ésta responsable de gestionar la interfaz haciendo uso de los recursos del entorno o, en el caso de ser una tarea adaptadora de artefacto, sólo sería responsable de la ubicación de la interfaz en el marco del entorno, debiendo ser construida por el artefacto.

Las tareas se formulan mediante modelos que tienen una parte común, relativa a su relación con el entorno *MDDE*, y una parte específica, relativa al mecanismo con el que se implementa la funcionalidad representada.

Existe un modelo abstracto común a todas las tareas *MDDE*, el cual formaliza:

- **La información requerida para su gestión desde la herramienta que la integra**, esto es, la información necesaria para su invocación y la que retorna como estatus resultante de su ejecución.
- **La información requerida para que su información asociada pueda ser gestionada desde los recursos del sistema**, esto es, para que la información de configuración sea visualizada y modificada en el marco de gestión de la herramienta y su estatus de ejecución sea visualizado también en él.

MDDE workbench

Se trata de la GUI de referencia que ha de poseer todo entorno MDDE para posibilitar la interacción del operador con la información contenida en el espacio de trabajo y con los procesos que se ejecutan en el entorno. La Figura 4.6 lo esquematiza. Como puede observarse, básicamente se compone de siete secciones:

- **Área superior**, constituida por las secciones 1 y 2.
- **Zona central**, constituida exclusivamente por la sección 3.
- **Región periférica**, constituida por las secciones 4, 5, 6 y 7. A su vez, cada sección está formada por uno o más *marcos de interacción*. Por *marco de interacción* se entiende un área gráfica que reúne un grupo fijo de visores y controles complementarios que implementan un mecanismo de interacción con el operador.

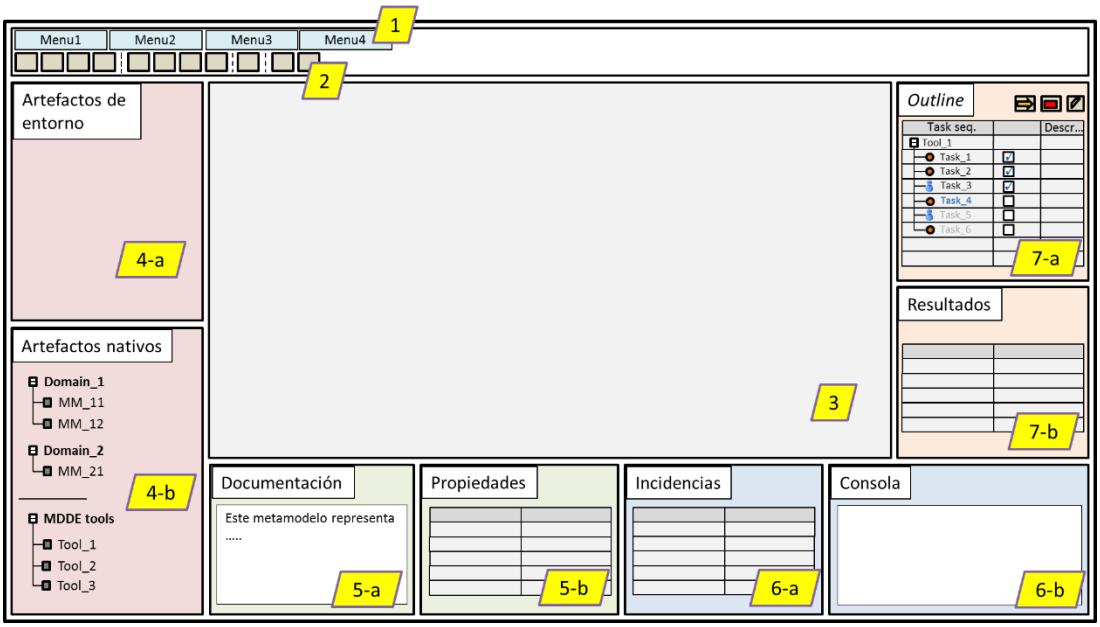


Figura 4.6 – Esquema del *MDDE workbench.*

A continuación se describen cada una de las secciones:

1. **Sección de gestión general del entorno**. Es una barra de menús que permiten gestionar globalmente el entorno. Se adopta de la plataforma tomada como base de los entornos.
2. **Sección de herramientas**. Es una barra de botones de herramientas que representan el más directo de los posibles métodos para el lanzamiento de herramientas. Cuando se pulsa un botón, se despliega un formulario que permite seleccionar la herramienta que se quiere lanzar.
3. **Sección para apertura de GUI de tarea interactiva**. La zona central del *MDDE workbench* se deja libre para despliegue de interfaces gráficas correspondientes a tareas interactivas, controladas por el operador.
4. **Sección de gestión de artefactos**. Presenta dos marcos de interacción de tipo explorador para organizar y acceder a la información disponible en el entorno, en forma de modelos, metamodelos, etc. Los artefactos se organizan mediante estructuras en árbol.
   a. **Navegador de *artefactos de entorno*.** Marco que permite la visualización, organización, selección y acceso a los *modelos y metamodelos de entorno*, esto es, aquellos que el usuario va produciendo durante sus sesiones de trabajo con el entorno (bien dedicadas al desarrollo de sistemas o bien al diseño de extensiones del propio entorno). Así pues, la información gestionada por medio de este marco es persistente y se encuentra almacenada en el espacio de trabajo asociado[60], organizada mediante el paradigma de contenedores anidados propio de los sistemas convencionales de ficheros.

      La selección que permite este marco es el punto de partida para la aplicación de herramientas sobre el artefacto seleccionado, posibilitando el acceso a la información contenida en él y su procesamiento.

[60] Si el sistema se cierra o cae por cualquier causa, la información contenida en este explorador está de nuevo disponible cuando se vuelve a abrir el entorno.

- Mediante selección contextual se ha de poder lanzar cualquiera de las herramientas existentes en el entorno que le sean aplicables. Éstas pueden ir desde diversos tipos de editores o de visores que permitan la presentación de la información contenida según una determinada vista hasta herramientas propiamente diseñadas según *MDDE*.
- Mediante selección directa (doble clic) se ha de presentar el modelo abierto en el editor establecido como editor por defecto.

b. **Explorador de *artefactos nativos***. Marco que permite la visualización, selección y acceso a los artefactos nativos del entorno. Puesto que éstos no residen en el espacio de trabajo, su organización se considera algo preestablecido y por tanto no modificable, de forma que este marco no está concebido para ello. De hecho, su propósito principal es únicamente posibilitar la inspección de tales elementos nativos, los cuales están caracterizados como recursos de sólo lectura y bajo este modo se realiza su apertura cuando son seleccionados (doble clic).

5. **Sección de exposición de información sobre elementos**. Presenta dos marcos para exponer información relativa al elemento seleccionado en cada momento en los marcos de la sección 3.

   a. **Documentación**. Marco que permite mostrar textualmente al operador información etiquetada como documentación complementaria asociada al elemento seleccionado. Además, en el caso de tratarse de un artefacto de entorno (no nativo), seleccionado por tanto en el marco 3-a, también permite al operador editar dicha información documental.

   b. **Propiedades**. Marco que permite mostrar de forma tabular las distintas propiedades de los elementos seleccionados en otros marcos, en particular las propiedades de elementos de modelos.

6. **Sección de exposición de información de la ejecución de herramientas**. Presenta dos marcos para exponer información procedente de la ejecución de las herramientas.

   a. **Incidencias**. Marco que permite mostrar de forma tabular las incidencias detectadas durante la ejecución de herramientas. Para cada incidencia, y en un conjunto de campos normalizados, se expresa información como por ejemplo la naturaleza y severidad (error, advertencia o recomendación) de la incidencia, la localización del elemento en el que se ha detectado y consejos de cómo puede ser subsanada.

   b. **Consola**. Marco que permite mostrar los avisos y mensajes textuales que las herramientas envían al operador durante su ejecución así como la introducción de información por parte de éste por línea de comandos.

      En principio, el motivo de la existencia de este marco es que, como se ha expuesto en los apartados anteriores, *MDDE* contempla que dentro de la secuencia de tareas de una herramienta existan tareas del tipo *adaptadora de artefacto externo*, cuya ejecución consiste en la invocación de una herramienta legada. En este caso, es muy probable encontrar herramientas originalmente diseñadas para ser utilizadas mediante línea de comandos y hacer uso de la salida estándar por consola como medio de comunicar resultados al usuario. Aunque adaptar el uso de tales herramientas desde un entorno

pueda invitar a reemplazar la salida en consola por otras formas de mayor riqueza visual para mostrar información, es muy posible que los usuarios acostumbrados a la salida estándar aún quieran disponer de ella. Por tanto, un entorno *MDDE* ha de dar soporte a esta posibilidad, a la cual da servicio este marco de interacción.

7. **Sección de gestión de herramientas**. Concebida para proporcionar al operador la capacidad de controlar la ejecución de una herramienta que ha sido previamente invocada, permitiéndole:
   - Visualizar la información asociada a la herramienta.
   - Establecer la configuración de la herramienta.
   - Llevar a cabo de forma controlada la ejecución de la herramienta.
   - Supervisar el estatus de ejecución que mantiene al ejecutarse.
   - Finalizar o abortar la ejecución de la herramienta.

   Esta zona ofrece dos marcos:

   a. **Esquema de herramienta (*tool outline*)**. Marco cuya función es mostrar la constitución de una herramienta, dando acceso a la información que tiene asociada y permitiendo controlar su ejecución. Al invocar el lanzamiento de una herramienta, se hace visible este marco (en caso de no encontrarse ya visible) 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 ésta se encuentra ejecutando y se vacía a su conclusión.

      La Figura 4.7 muestra los elementos de interacción que ofrece este marco.

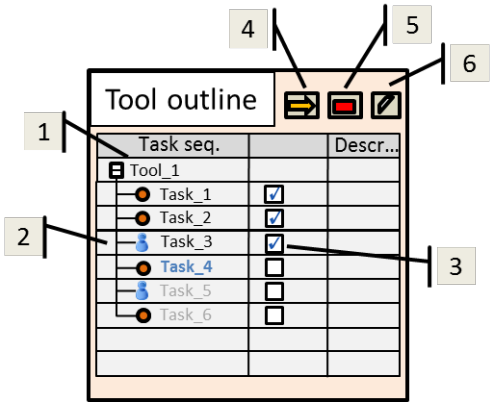


Figura 4.7 – Elementos de interacción del marco *Tool outline*

      1. **Ítem raíz**. Representa la herramienta. Su selección puebla el marco *Propiedades* con sus parámetros de configuración, permitiendo modificar sus valores.
      2. **Ítems hoja**. Representan las tareas que componen la herramienta, indicando si son atómicas o interactivas. Éstos ítems muestran si, según el estado de ejecución de la herramienta, las tareas están o no habilitadas para ser ejecutadas.

         La selección de una tarea puebla el marco *Propiedades* con sus parámetros de configuración, permitiendo modificar sus valores si se trata de parámetros reconfigurables.

3. ***Check box***. La casilla de verificación que existe junto a cada tarea indica si la tarea ha sido ya completada con éxito.
4. **Botón de ejecución**. Botón para ejecutar la tarea que esté seleccionada, siempre y cuando se encuentre habilitada.
5. **Botón de conclusión**. Botón para dar por terminada una tarea interactiva.
6. **Botón de finalización**. Botón que finaliza la ejecución de la herramienta y vacía el marco *outline*.

b. **Visor de resultados**. Este marco permite mostrar de forma tabular los resultados generados por la ejecución de una herramienta.

4.1.2 Metamodelado de la parte conceptual del *framework* de *MDDE*

En este apartado se presenta el metamodelo MDDE mediante el que se formalizan los aspectos estructurales del *framework* de *MDDE*.

Objetivos del diseño

Para el metamodelo MDDE se ha optado por una estrategia de diseño que permite especificar un entorno *MDDE* mediante la formulación conjunta de varios modelos, todos conformes a él:

- **Modelos de formulación de las herramientas presentes en el entorno**, encapsulando las tareas constituyentes.
- **Modelos de formulación de los *tipos de tareas* definidos en el entorno**. Por *tipo de tarea* se entiende un ente parametrizado (configurable) cuya realización específica (asignación de valores a sus parámetros de configuración) da lugar a una tarea concreta.

  Así, las tareas contenidas en los modelos de herramientas son realizaciones de los *tipos de tareas* formulados mediante estos modelos.
- **Modelo del entorno**. Se trata del modelo que representa propiamente al entorno, pero cuyo papel se reduce simplemente a ser un aglutinador de los modelos anteriores.

La Figura 4.8 muestra la especificación de entornos *MDDE* en base a tales modelos.

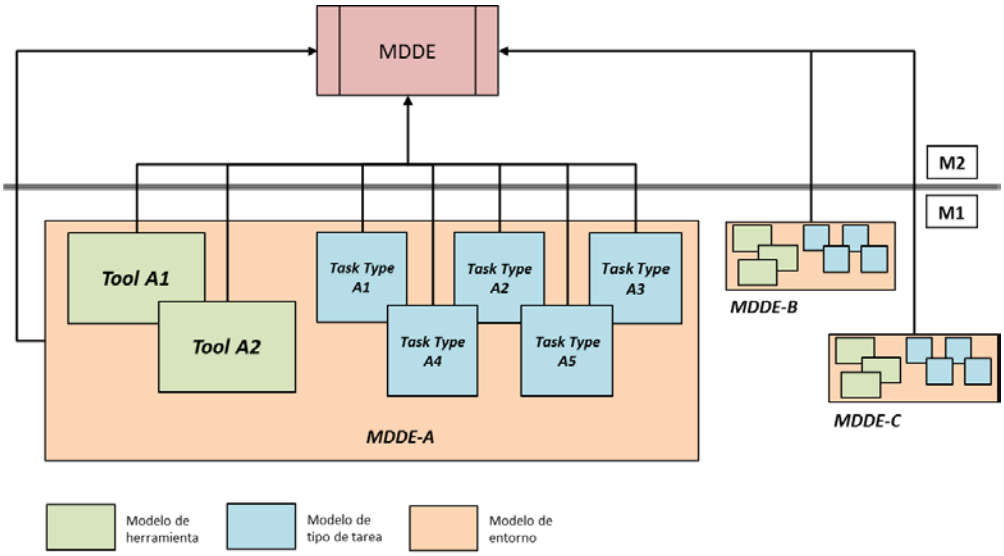


Figura 4.8 – Especificación de entornos *MDDE* mediante modelos

Aunque en un principio se optó por únicamente el primer conjunto de modelos (junto al modelo aglutinador), la posible participación de la misma tarea (o de tareas análogas) en diferentes herramientas provocó la decisión de buscar la extracción de la semántica básica de las tareas fuera de los modelos de las herramientas. Esto dio lugar a emplear en el diseño del metamodelo el patrón *instancia-descriptor* y consecuentemente al segundo conjunto de modelos. En los siguientes apartados, dedicados a la estructura de clases del metamodelo, se profundiza en esta dualidad.

Núcleo del metamodelo

La Figura 4.9 muestra el núcleo del metamodelo **MDDE**. En función de todo lo expuesto anteriormente, la semántica de las clases mostradas es evidente:

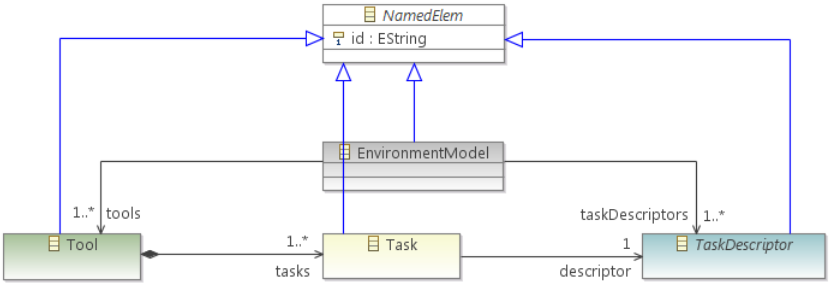


Figura 4.9 – Núcleo del metamodelo MDDE

- **`Tool`**. Representa el concepto de herramienta o proceso.

  Su asociación `tasks` permite que una instancia de `Tool` albergue la secuencia de sus tareas constituyentes, instancias de la clase `Task`.

- ***`TaskDescriptor`***. Representa el concepto de *tipo de tarea*.

- **`Task`**. Representa el concepto de tarea, como realización concreta de un *tipo de tarea* (instancia de la clase *`TaskDescriptor`* y apuntada mediante la referencia `descriptor`).

- **`EnvironmentModel`**. Representa el concepto de entorno.

  Una instancia de esta clase referencia el conjunto de herramientas y el conjunto de *tipos de tareas* que conforman la especificación de un entorno *MDDE* a través de las asociaciones `tools` y `taskDescriptors`, respectivamente.

Las clases *`TaskDescriptor`* y `Task` reflejan el hecho de haber empleado el patrón *instancia-descriptor* en el diseño del metamodelo. Por tanto, aunque ambas sean clases del metamodelo, no están a igual nivel de abstracción, pues representan respectivamente los conceptos de *tipo de tarea* y de *realización concreta de un tipo de tarea*.

Por último, aunque pueda parecer que el metamodelo presenta una estructura convencional, con la clase `EnvironmentModel` desempeñando el rol de clase contenedor principal, sus asociaciones `tools` y `taskDescriptors` no exhiben carácter de composición. Esta clase simplemente da soporte al modelo aglutinador del resto de modelos. Así, las clases `Tool` y *`TaskDescriptor`* también poseen rol de contenedor principal y cada modelo de herramienta y cada modelo de *tipo de tarea* cuenta respectivamente con una instancia suya en su raíz.

Metamodelado de tipos de tarea

Un *tipo de tarea* se formula mediante un modelo cuyo contenedor principal es una instancia de la clase *`TaskDescriptor`*. La Figura 4.10 se centra en esta clase y en cómo definir la configurabilidad de un *tipo de tarea*.

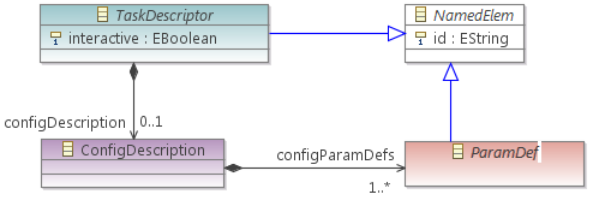


Figura 4.10 – Clase *`TaskDescriptor`*

Básicamente, un descriptor de tarea especifica un identificador y si se trata de un *tipo de tarea* con carácter atómico o interactivo (atributo `interactive`). Además, opcionalmente y a través de la asociación `configDescription`, contiene un submodelo que encapsula las opciones de configuración del *tipo de tarea* en cuestión. Estas opciones se formulan mediante instancias de tipo *`ParamDef`*, clase abstracta que representa el concepto de definición de un parámetro de configuración para un *tipo de tarea*.

El metamodelo **MDDE** define una jerarquía de subclases que especializan a *`ParamDef`*. Mediante ellas se pueden definir parámetros cuyo posible valor sea de tipo entero, real, booleano, textual, enumerado o del muy útil tipo `LocatorParamDef`. Por medio de este último se pueden formular parámetros representativos de rutas donde localizar modelos.

La Figura 4.11 muestra este conjunto de subclases para definición de parámetros de configuración.

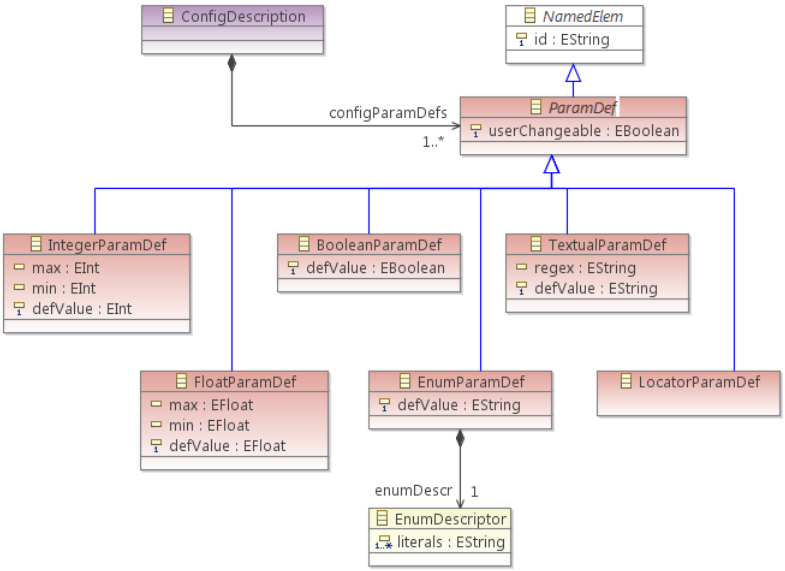


Figura 4.11 – Subclases de *`ParamDef`*

En el caso de definir parámetros numéricos (mediante instancias de las clases `IntegerParamDef` y `FloatParamDef`), pueden definirse límites superior e inferior (atributos `max` y `min`) que acoten el rango de valores permitidos y, análogamente, en el caso de definir parámetros textuales (mediante instancias de `TextualParamDef`) puede especificarse una expresión regular (atributo

regex) que determine los valores válidos que puede adoptar tal parámetro. Por último, al definir un parámetro mediante una instancia de estas subclases de *ParamDef*, se ha de especificar un valor por defecto para él (mediante el atributo `defValue`), del tipo primitivo correspondiente.

La Figura 4.12 muestra las subclases de *TaskDescriptor*.

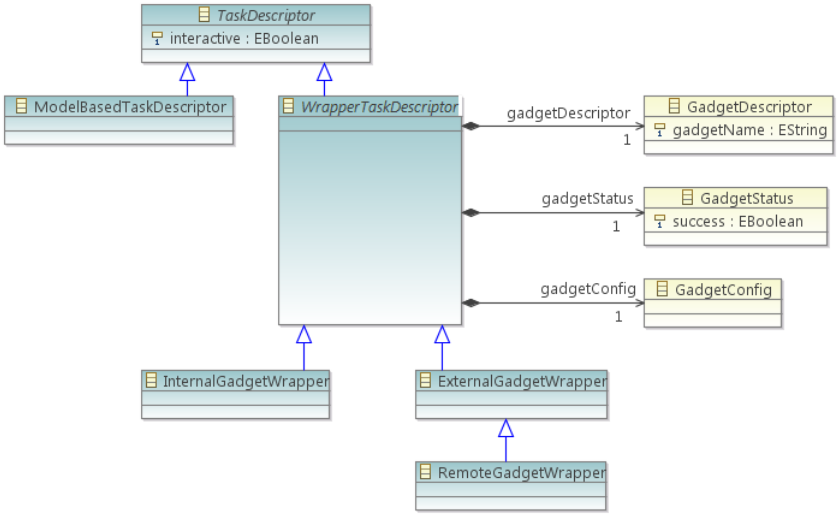


Figura 4.12 – Subclases de *TaskDescriptor*

Metamodelado de herramientas

Una herramienta se formula mediante un modelo cuyo contenedor principal es una instancia de la clase `Tool`. La Figura 4.13 muestra la estructura de esta clase.

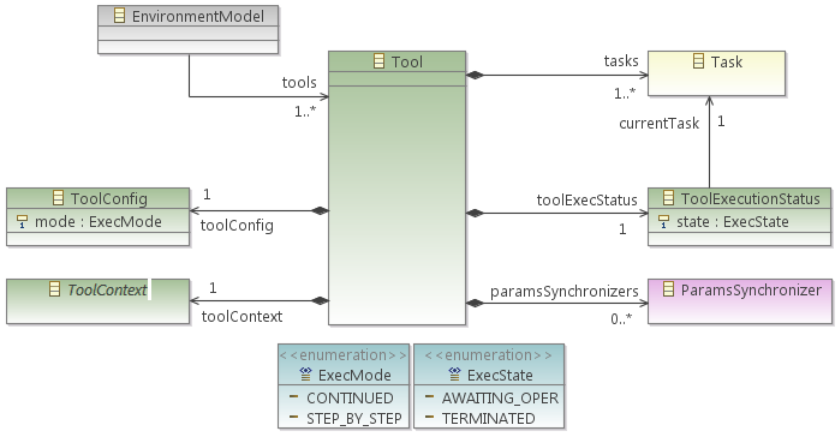


Figura 4.13 – Clase *Tool*

Además de la asociación `tasks` ya conocida, esta clase define otras cuatro agregaciones más:

- **`toolContext`**. A través suyo, una herramienta contiene un objeto de tipo *ToolContext*, representativo de su contexto de aplicación.
- **`toolConfig`**. A través suyo, una herramienta contiene un objeto de tipo `ToolConfig`, representativo del submodelo que encapsula las opciones de configuración de la herramienta.

  En la versión actual del metamodelo, esto se reduce únicamente al modo de ejecución (continuado o por pasos).

- **`toolExecStatus`**. A través suyo, una herramienta contiene un objeto de tipo `ToolExecutionStatus`, representativo del submodelo que encapsula la información de estatus que ha de mantener la herramienta durante su ejecución.

  En la versión actual del metamodelo, esto se reduce únicamente al estado de ejecución y a la tarea en curso en cada momento.

- **`paramsSynchronizers.`** A través suyo, una herramienta contiene un conjunto de objetos de tipo `ParamsSynchronizer`. Su función es permitir la asignación sincronizada de valores a parámetros de configuración por parte de las tareas constituyentes (en la siguiente sección se explica más en detalle este mecanismo).

Metamodelado de tareas (realizaciones de tipos de tarea)

En la formulación del modelo de una herramienta se ha de modelar su secuencia de tareas constituyentes mediante instancias de la clase `Task` que entren a formar parte de la asociación `Tool.tasks`, donde cada instancia de `Task` representa una realización concreta del *tipo de tarea* descrito mediante el objeto *`TaskDescriptor`* referenciado a través de la asociación `Task.descriptor` (ver Figura 4.9).

Por realización concreta de un *tipo de tarea* se entiende el resultado de asignar un valor a cada uno de los parámetros de configuración definidos por el correspondiente descriptor, aunque en el caso de considerarse apto el valor por defecto estipulado en la definición del parámetro, no sería necesario realizar asignación explícita.

La Figura 4.14 muestra la relación entre las clases *`TaskDescriptor`* y `Task` en base a la capacidad de definición de parámetros de configuración por parte de la primera y a la capacidad de asignación de valores por parte de la segunda.

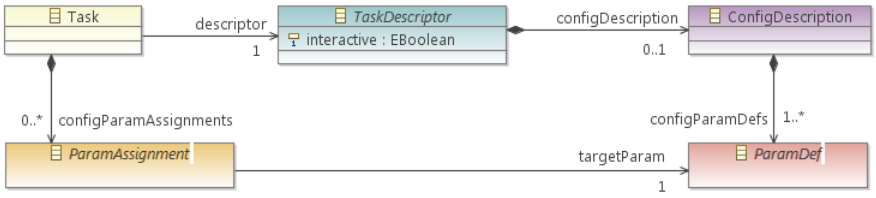


Figura 4.14 – Definición y asignación de parámetros

La clase `Task` define la asociación `configParamAssignments` que permite a una tarea concreta alojar una serie de objetos (instancias *`ParamAssignment`*) representativos de asignaciones de valores a parámetros. La definición del parámetro correspondiente (objeto *`ParamDef`*) viene determinado por la referencia `targetParam` y, evidentemente, debe ser un parámetro definido en el seno del descriptor correspondiente a la tarea.

El siguiente apartado profundiza en la asignación de valores a parámetros de configuración por parte de las tareas constituyentes de una herramienta.

Asignación de parámetros

En una tarea se consideran dos formas de asignar valores a sus parámetros de configuración:

a. **Asignación directa**. Se declara explícitamente un valor concreto a ser asignado, respetando naturalmente el tipo del parámetro (la subclase de *`ParamDef`* empleada para su definición).

Mediante esta técnica, la asignación que una tarea realiza sobre un parámetro es independiente de cualquier otra asignación llevada a cabo en el conjunto de tareas del que forma parte.

b. **Asignación sincronizada**. No siempre la asignación de valores a parámetros de configuración es un caso de *asignación directa*. En algunas situaciones, distintos parámetros de configuración (aunque del mismo tipo) definidos independientemente por diversos descriptores de tarea han de adoptar valores sincronizados en el contexto de una herramienta determinada. Esto significa que el valor que se asigna a cada uno de ellos por las correspondientes instancias `Task` en la herramienta ha de ser el mismo, de manera que si posteriormente uno de ellos es modificado por el operador, los demás han de variar solidariamente.

Para dar soporte a estas técnicas de asignación, la clase abstracta *`ParamAssignment`* se especializa en las dos subclases concretas que se muestran en la Figura 4.15.

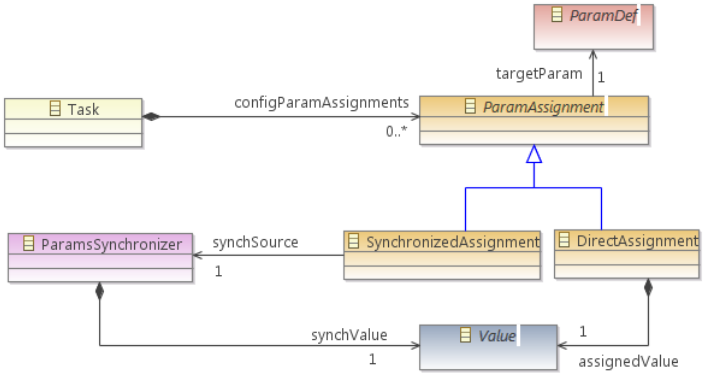


Figura 4.15 – Tipos de asignación de valores a parámetros

- **`DirectAssignment`**. Clase que soporta la asignación directa de valores a parámetros. Por medio de la asociación `assignedValue` se permite un objeto de tipo *`Value`* mediante el que se formula el valor que se asigna al parámetro de configuración correspondiente.
- **`SynchronizedAssignment`**. Clase que, en colaboración con la clase `ParamsSynchronizer`, soporta la asignación sincronizada de parámetros. La clase `ParamsSynchronizer` juega un rol similar a `DirectAssignment` en cuanto a que declara la contención de una instancia *`Value`* que sirve para formular un valor, en este caso valor común compartido por todas las asignaciones que se hallan sincronizadas mediante el objeto sincronizador apuntado por `SynchronizedAssignment.synchSource`.

`Value` es también una clase abstracta cuya jerarquía de subclases concretas se muestra en la Figura 4.16. Como puede observarse, existe un paralelismo lógico entre las subclases de *`ParamDef`* (ver Figura 4.11) y las subclases de *`Value`*. Obviamente, cuando se modela una herramienta, esta correspondencia ha de ser respetada al formular la asignación de valores a los parámetros de configuración.

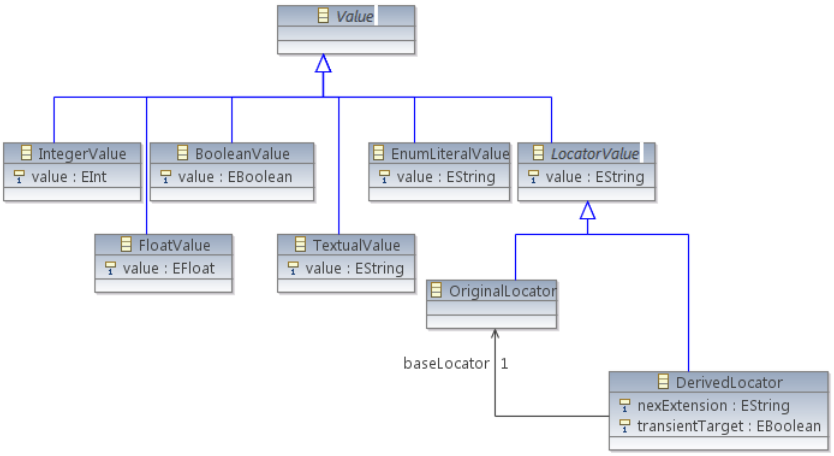


Figura 4.16 – Subclases de *Value*

4.2 Herramientas y recursos para diseño de los entornos

Las herramientas y recursos que se presentan en esta sección están orientadas a automatizar o asistir al diseñador de entornos en los procesos de incorporación a dicho entorno de los recursos y funcionalidades de los que va a disponer el ingeniero final que lo utiliza para desarrollar un proyecto software. Básicamente, se han incluido funcionalidades relativas al lanzamiento de herramientas y gestión de su flujo.

4.2.1 *MDDE extension workbench*

MDDE define una variante de su *workbench* original orientada a los usuarios diseñadores de entornos específicos. Esta variante toma el nombre de *MDDE extension workbench* y su *layout* simplifica el de la versión original, pues prescinde de los marcos *Outline*, *Resultados* y *Consola*. La Figura 4.17 esquematiza esta variante.

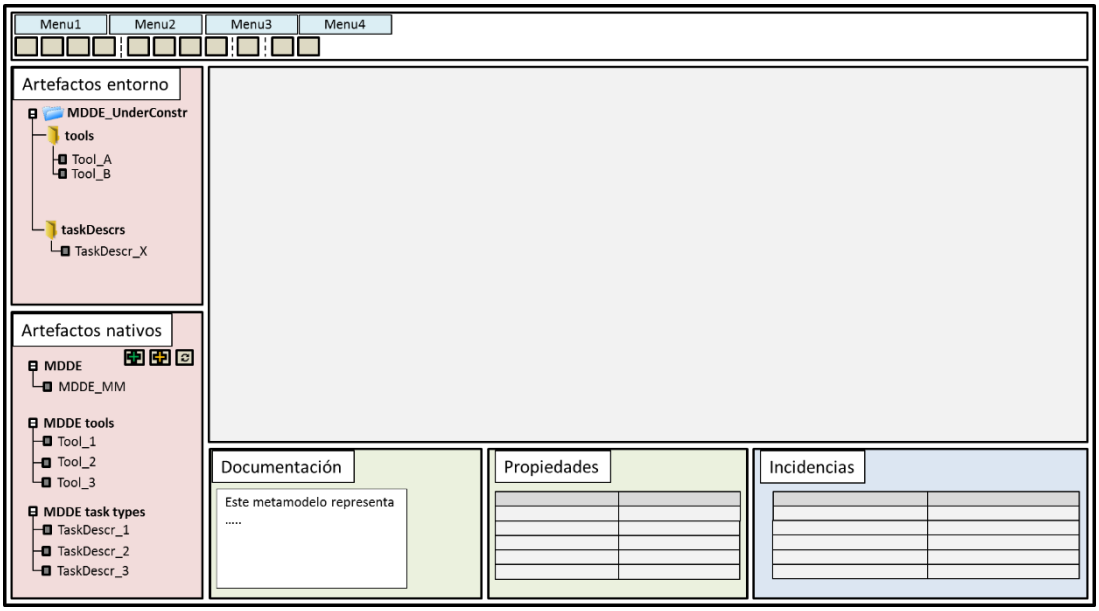


Figura 4.17 – Esquema del *MDDE extension workbench*

Además, los contenidos de los marcos *Explorador de artefactos de entorno* y *Navegador de artefactos nativos* se ven modificados, ya que ahora el ámbito de desarrollo es el propio dominio

MDDE y al usuario le resultan de interés los tipos de tarea definidos en el entorno, de los cuales puede crear instancias en las nuevas herramientas que diseñe. Asimismo, su labor también puede consistir en diseñar nuevos tipos de tarea.

La Figura 4.18 y la Figura 4.19 muestran el cambio que experimentan estos marcos al pasar del *MDDE workbench* original (izqda.) al *MDDE extension workbench* (dcha.). Como puede observarse, en el marco *Explorador de artefactos de entorno* se ocultan cualesquiera artefactos propios de sesiones de trabajo relativas al diseño de sistemas y en su lugar se muestra una sencilla estructura de contenedores en la cual almacenar los modelos de herramientas y tipos de tareas que el diseñador de entornos formule.

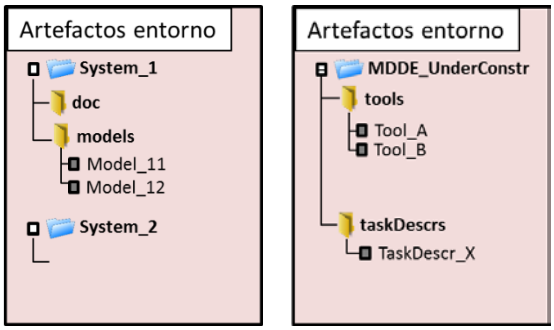


Figura 4.18 – Marco *Explorador de artefactos de entorno*

En cuanto al marco *Navegador de artefactos nativos*, se ocultan cualesquiera artefactos propios del ámbito específico de desarrollo de sistemas y en su lugar se muestra el metamodelo MDDE. Además, se muestran los tipos de tareas definidos en el entorno. Adicionalmente, ahora el marco está equipado con controles para añadir una nueva herramienta, añadir un nuevo tipo de tarea o refrescar la visualización.

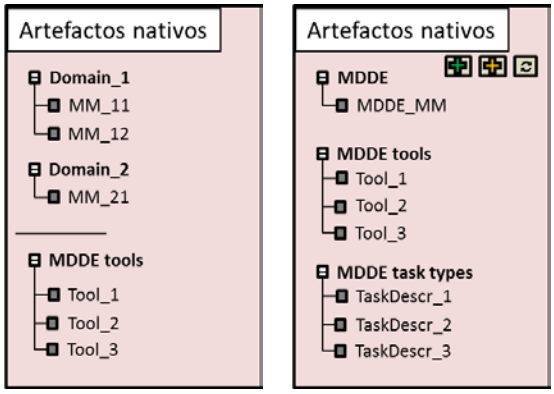


Figura 4.19 – *Marco navegador de artefactos nativos*

4.2.2 Diseño de nuevas herramientas

Se proporciona al usuario diseñador de entornos el asistente *NewTool* para creación de nuevas herramientas (modelos conformes al metamodelo MDDE con una instancia de la clase `Tool` como contenedor principal). El empleo de tal asistente sigue el siguiente protocolo:

1. El usuario lo invoca mediante el botón *Añadir herramienta* disponible en el marco *Navegador de artefactos nativos*.
2. Se presenta un cuadro de diálogo para introducir el nombre de la herramienta a ser desarrollada.

3. Se inicializa en memoria un modelo de herramienta *MDDE*. El modelo contendrá la instancia `Tool` contenedor principal adecuadamente inicializada y dos instancias de la clase `Task`.
    i. La primera, identificada como *Task_1*, no se encuentra asignada a ningún descriptor de tarea.
    ii. La segunda, identificada como *Termination*, es de tipo finalización de ejecución de herramienta.
4. Se persiste en el espacio de trabajo el modelo generado, creando un fichero `toolName.tool.mdde` donde `toolName` es el nombre introducido por el usuario.

   El almacenamiento se realiza en la ruta `workspace/MDDE_UnderConstr/tools`, donde `workspace` es la ruta del espacio de trabajo y la estructura de contenedores `MDDE_UnderConstr/tools` se crea si no existe.
5. Se abre el modelo en el área de edición para que el operador proceda a completar su construcción, esto es, completar la formulación de la primera tarea y añadir más si procede.

4.2.3 Diseño de nuevos tipos de tareas

Se proporciona al usuario diseñador de entornos el asistente *NewTaskDescriptor* para creación de nuevos tipos de tareas (modelos conformes al metamodelo MDDE con una instancia de la clase `TaskDescriptor` como contenedor principal). El empleo de tal asistente sigue el siguiente protocolo:

1. El usuario lo invoca mediante el botón *Añadir descriptor de tarea* disponible en el marco *Navegador de artefactos nativos*.
2. Se presenta un cuadro de diálogo para introducir el nombre del descriptor a desarrollar.
3. Se inicializa en memoria un modelo de descriptor de tarea *MDDE*. El modelo contendrá la instancia `TaskDescriptor` contenedor principal adecuadamente inicializada.
4. Se persiste en el espacio de trabajo el modelo generado, creando un fichero `taskDescriptorName.taskdescr.mdde`, donde `taskDescriptorName` es el nombre introducido por el usuario. El almacenamiento se realiza en la ruta `workspace/MDDE_UnderConstr/taskDescriptors`, donde `workspace` es la ruta del espacio de trabajo y la estructura de contenedores `MDDE_UnderConstr/taskDescriptors` se crea si no existe.
5. Se abre el modelo en el área de edición para que el diseñador proceda a completar su construcción.

4.2.4 Registro de las extensiones desarrolladas

Puesto que el completado de los modelos tanto de herramientas como de tipos de tarea puede prolongarse durante un número indefinido de sesiones de trabajo, éstos permanecen persistidos en el espacio de trabajo, sin considerarse que representen extensiones *MDDE* incorporadas de facto al entorno.

Cuando se considera que la formulación de un modelo de herramienta o descriptor de tarea ha sido satisfactoriamente completada, el siguiente paso natural es su incorporación al entorno. Para ello, se definen en *MDDE* las funcionalidades *Registrar herramienta* y *Registrar tipo de tarea*, accesibles respectivamente mediante selección contextual de un modelo de herramienta

o tipo de tarea en la vista *Explorador de artefactos de entorno*. El registro de estos modelos implica una fase de verificaciones preliminares (relativas a conformidad respecto al metamodelo MDDE y cumplimiento de restricciones) que, en caso de ser superada, conducen al registro propiamente dicho. Esto significa el traslado del modelo desde su ubicación provisional en el espacio de trabajo a la ubicación establecida para las extensiones *MDDE*, con lo que desaparece del marco *Explorador de artefactos de entorno* y aparece en el marco *Navegador de artefactos nativos*. A partir de entonces, en caso de tratarse de una herramienta, ya se encuentra disponible para su uso por parte de un diseñador de sistemas o, en caso de tratarse de un descriptor de tarea, para su uso por el propio diseñador de entornos.

4.3 Soporte funcional

De lo expuesto en las secciones anteriores de este capítulo se desprende una serie de aspectos a los que ha de dar soporte funcional aquella plataforma elegida como base sobre la que construir entornos *MDDE*.

Por un lado, dentro de la sección 4.1 se ha expuesto el *MDDE workbench*, la GUI normalizada con la que todo entorno *MDDE* ha de estar equipado. En lugar de realizar su implementación desde cero, es conveniente que la plataforma de soporte escogida proporcione una infraestructura básica que permita su construcción a bajo coste, a ser posible mediante la reutilización de componentes adaptables.

Por otro lado, el análisis de las funcionalidades a nivel de usuario final que cada entorno *MDDE* ha de implementar conduce a identificar otras de más bajo nivel, relacionadas principalmente con el manejo de modelos, en base a las que se implementan las primeras. El soporte a tales acciones básicas es también responsabilidad de la plataforma base, la cual ha de posibilitar su implementación mediante el uso de la infraestructura nativa que ofrezca. Se han identificado las siguientes acciones:

- Representación en memoria de modelos.
- Carga en memoria de un modelo persistido.
- Ejecución de transformaciones M2M.
- Lanzamiento programático de la ejecución de transformaciones M2M.
- Inicialización de nuevos modelos.
- Construcción programática de un modelo (activo).
- Almacenamiento persistente de un modelo activo.

Las siguientes subsecciones abordan estos requisitos que *MDDE* exige a una plataforma para ser empleada como base. Además, en cada caso se presenta la manera en que Eclipse/EMF facilita su implementación. Cabe resaltar que su empleo como plataforma de soporte supone que todas las tareas de programación se realizan en lenguaje Java.

4.3.1 Implementación del *MDDE workbench*

El *workbench* de Eclipse representa una base idónea sobre la que implementar el *MDDE workbench*. Se presenta a continuación un breve apartado exponiendo sus nociones básicas y después se expone cómo puede construirse el *workbench* de los entornos *MDDE* en base a él.

El workbench de Eclipse

Se define como el escritorio del entorno de desarrollo y proporciona un paradigma común para la creación y gestión de los recursos del espacio de trabajo, así como para su navegación. Se presenta en forma de *ventana de workbench* (en todo momento pueden existir abiertas varias de ellas) y su propósito es lograr una integración continua de herramientas manteniendo una expansión controlada.

El *workbench* consta de componentes visuales llamados genéricamente *partes* y proporciona numerosas operaciones para trabajar con ellos. Existen dos tipos de *partes* del *workbench*: *vistas* y *editores*. Dentro de una ventana del *workbench*, los editores se alojan en la zona central (*área de editor*) y las vistas se disponen a su alrededor. Cualquier *parte* puede encontrarse tanto activa como inactiva, aunque sólo una como máximo puede estar activa en un momento dado. Ésta se distinguiría por poseer su barra de título resaltada y sería el destino de operaciones como cortar, copiar y pegar. La *parte* activa también determina los contenidos de la línea de estado.

Las vistas se usan principalmente para navegar por una lista o jerarquía de información (como por ejemplo los recursos en el espacio de trabajo) o visualizar propiedades relativas al editor activo. Las modificaciones hechas en una vista son salvadas inmediatamente. Espacialmente, las vistas pueden aparecer solas o apiladas de manera multipestaña y durante el uso normal del *workbench* pueden ser movidas, redimensionadas, cerradas y abiertas de nuevo.

El mecanismo de expansión de Eclipse proporciona el punto de extensión *org.eclipse.ui.views* que permite definir declarativamente extensiones en forma de nuevas vistas. En código, cada nueva vista ha de estar respaldada por una clase (Java) que implemente la interfaz `IViewPart`.

Los editores ofrecen presentación visual (textual o gráfica) de los recursos del espacio de trabajo, posibilitando navegar por sus contenidos y permitiendo su modificación. Espacialmente y siempre dentro del *área de editor*, los editores pueden aparecer apilados (situación por defecto) o también adyacentes, de forma que puedan verse simultáneamente sus contenidos. Sin embargo, aunque sea posible abrir simultáneamente un número arbitrario de editores, sólo uno puede encontrarse activo. Dado un editor activo, las barras de menús y de herramientas de la ventana de *workbench* proporcionan operaciones aplicables al editor.

Análogamente al caso de las vistas, el punto de extensión *org.eclipse.ui.editors* permite definir nuevos editores, cada uno respaldado por una clase que implemente a `IEditorPart`.

El tercer tipo de elemento (aunque no propiamente componente) del *workbench* son las perspectivas. Una perspectiva es una selección de vistas y una descripción de su posicionamiento en una ventana del *workbench*, controlando además qué aparece en ciertos menús y barras de herramientas. Tal descripción de posicionamiento se articula en torno al área de editor y el propósito de definir una perspectiva es facilitar ciertos tipos de tareas o trabajo con un tipo concreto de recursos.

Cada producto determina inicialmente qué perspectiva se muestra por defecto, aunque lo habitual en una sesión de trabajo es permutar entre perspectivas frecuentemente. Así, una ventana de *workbench* puede presentar más de una perspectiva abierta (aunque sólo una activa en cada momento) y la apertura de una nueva perspectiva puede hacerse en la misma ventana o en una nueva. Dentro de una ventana, cada perspectiva puede tener un conjunto diferente de vistas pero todas ellas comparten el mismo conjunto de editores.

El punto de extensión *org.eclipse.ui.perspectives* permite definir nuevas perspectivas, cada una respaldada por una clase que implemente a `IPerspectiveFactory`.

MDDE workbench en base al workbench de Eclipse

El empleo de Eclipse como plataforma de soporte permite implementar el concepto *marco de interacción* propio del *MDDE workbench* mediante el concepto *vista* propio del *workbench* de Eclipse. En algunos casos es incluso posible emplear vistas predefinidas y en otros se ha de diseñar la vista en cuestión.

La Tabla 4-1 recoge los marcos definidos en el *MDDE workbench* cuya funcionalidad queda perfectamente cubierta mediante vistas predefinidas en Eclipse.

Tabla 4-1 – Marcos de interacción cubiertos mediante vistas Eclipse predefinidas.

Marco	Vista
Navegador de artefactos de entorno	***Project Explorer***. Vista concebida para presentar jerárquicamente, en formato de árbol, los recursos del espacio de trabajo, permitiendo su selección para realizar acciones como renombrarlos, refrescarlos, moverlos, copiarlos, eliminarlos, compararlos, exportarlos, crear nuevos recursos o abrir ficheros para su edición.
Propiedades	***Properties***. Vista concebida para mostrar propiedades relativas a un cierto ítem seleccionado, como por ejemplo, un recurso en la vista *Project Explorer* o un elemento de un modelo abierto en un editor. Éstas se presentan en forma de par nombre-valor y la vista proporcionas diversas posibilidades, como mostrarlas agrupadas por categoría, filtrar propiedades avanzadas o restaurar una propiedad a su valor por defecto.
Consola	***Console*.** Vista concebida para mostrar la salida estándar por consola. Esta vista puede albergar diversas páginas simultáneamente y permite al usuario permutar entre ellas. Un *plug-in* puede optar por escribir sobre una página de consola ya registrada o contribuir su propia página. Una página de consola viene representada por una instancia de la interfaz `org.eclipse.ui.console.IConsole` que una vez creada ha de ser conectada a la vista *Console*.
Incidencias	***Problems***. Vista concebida para mostrar en forma tabular los problemas (errores, advertencias, etc.) aparecidos en los recursos que el trabajo habitual en el *workbench* suele provocar. Estos problemas se registran automáticamente en esta vista, donde por defecto son agrupados por severidad, aunque también pueden serlo por tipo o incluso no ser agrupados. La vista puede configurarse para mostrar únicamente los problemas asociados a un recurso o grupo de recursos en particular. También admite añadir diversos filtros (tanto aditivos como exclusivos) que pueden habilitarse o deshabilitarse a voluntad. Una de las funciones más útiles de esta vista es que a partir de la selección de uno de los problemas mostrados, el recurso asociado se abre en el editor apropiado a la altura de la línea de código problemática.

El empleo de vistas predefinidas requiere un esfuerzo variable en cuanto a codificación. Por ejemplo, utilizar la vista *Project Explorer* típicamente sólo implica su posicionamiento y apertura, operaciones inmediatas mediante su ID. Sin embargo, en otros casos es necesario realizar una adaptación de mayor envergadura. A modo de ejemplo, el Código 4.1 muestra cómo crear una

página de consola utilizando la clase `MessageConsole` que implementa la interfaz `IConsole`[61].

```
MessageConsole mddeConsole = new MessageConsole("MDDE", null);
IConsoleManager consoleMngr = ConsolePlugin.getDefault().getConsoleManager();
consoleMngr.addConsoles(new IConsole[]{mddeConsole});
```

Código 4.1 – Creación de una página de consola.

Una vez que se ha creado una consola, existe más de una vía para escribir en ella. El Código 4.2 localiza la consola creada y escribe abriendo un flujo de salida sobre ella.

```
IConsoleManager consoleMngr = ConsolePlugin.getDefault().getConsoleManager();
MessageConsole mddeConsole;
for (IConsole console : consoleMngr.getConsoles()) {
   if (console.getName().equals("MDDE")) {
      mddeConsole = (MessageConsole)console;
   }
}
MessageConsoleStream mcStream = mddeConsole.newMessageStream();
mcStream.println("Hello");
```

Código 4.2 – Escritura en consola

Crear una página de consola y escribir sobre ella no hace visible la vista *Console*. Ésta ha de ser abierta mediante el modo habitual de abrir vistas, empleando el API del *workbench*. Además, hay que tener en cuenta que la vista *Console* puede contener varias consolas diferentes, cada una proporcionada por un *plugin*. Por tanto, para mostrar una determinada consola no basta con abrir la vista sino que también hay que establecer explícitamente que la apertura se realice visualizando la consola en cuestión. El Código 4.3 abre la vista *Console* mostrando la consola MDDE.

```
import org.eclipse.ui.IViewPart;
import org.eclipse.ui.IWorkbenchPage;
...

String consoleViewID = "org.eclipse.ui.console.ConsoleView";

/* I get access to the "Console" view through the workbench infrastruture.
 * If it is not visible, I open it */
IWorkbenchPage wPage = PlatformUI.getWorkbench().
                                  getActiveWorkbenchWindow().getActivePage();
IViewPart consoleViewPart = wPage.findView(consoleViewID);
if (!wPage.isPartVisible(consoleViewPart)) {
   wPage.showView(consoleViewID);
}
IConsole mddeConsole = ...;
((IConsoleView)consoleViewPart).display(mddeConsole);
```

Código 4.3 – Apertura de consola

La Tabla 4-2 recoge el resto de marcos de interacción definidos, cuya funcionalidad se implementa por medio de vistas desarrolladas específicamente. Para cada una se indica su ID y la clase con el código Java de soporte. Asimismo, se indica el elemento GUI (control o visor) de entre los proporcionados por Eclipse que es posible emplear para dar cuerpo a cada vista en base a la definición del correspondiente marco de interacción a ser implementado.

[61] Las clases `MessageConsole` y `ConsolePlugin` así como la interfaz `IConsoleManager` también pertenecen al paquete `org.eclipse.ui.console`.

Tabla 4-2 – Marcos de interacción implementados mediante vistas específicamente diseñadas.

Marco	Vista
Explorador de *artefactos nativos*	***TechExplorer*** ▪ extension ID: **`es.unican.istr.mdde.views.TechExplorer`** ▪ clase: **`TechExplorer.java`** ▪ *TreeViewer*
Documentación	***Documentation*** ▪ extension ID: **`es.unican.istr.mdde.views.Documentation`** ▪ clase: **`Documentation.java`** ▪ *Text*
Esquema de herramienta	***ToolOutline*** ▪ extension ID: **`es.unican.istr.mdde.views.ToolOutline`** ▪ clase: **`ToolOutline.java`** ▪ *TableTreeViewer*
Visor de resultados	***Results*** ▪ extension ID: **`es.unican.istr.mdde.views.Results`** ▪ clase: **`Results.java`** ▪ *TableViewer*

Por último, la implementación del *MDDE workbench* en base al *workbench de Eclipse* posibilita cumplir lo estipulado en la especificación del primero en cuanto a la integración de GUIs correspondientes a tareas interactivas, esto es, su inclusión en la zona central del *workbench*. Para ello, las GUIs en cuestión adoptan la forma de editores.

4.3.2 Representación en memoria de modelos (conformes a un cierto metamodelo de dominio)

Las actividades llevadas a cabo durante la ejecución de las herramientas de un entorno *MDDE* son básicamente operaciones de procesamiento de modelos. Para su ejecución es necesario que la tecnología de la plataforma de soporte permita la existencia en memoria de representaciones de los modelos. El empleo de Eclipse/EMF como plataforma cubre perfectamente esta necesidad.

Soporte en Eclipse/EMF

Dado un cierto metamodelo, EMF permite obtener una implementación Java que posibilita la existencia en memoria de modelos conformes a tal metamodelo, como conjunto de instancias de las clases Java generadas. El código producido se organiza en tres paquetes, cuyos nombres se derivan del nombre del metamodelo:

- **`metamodelName`**. Contiene una interfaz por cada clase del metamodelo, adoptando el mismo nombre. Además, contiene dos interfaces más, relativas al metamodelo en su conjunto:
  - `MetamodelNamePackage`
  - `MetamodelNameFactory`
- **`metamodelName.impl`**. Contiene una clase por cada interfaz del paquete anterior. El nombre de cada una coincide con el nombre de la correspondiente interfaz, añadiendo el sufijo "`Impl`".

- **`metamodelName.util`**. Contiene dos clases de utilidad:
  - `MetamodelNameAdapterFactory`
  - `MetamodelNameSwitch`

La representación propiamente dicha de modelos en memoria es soportada por el conjunto de interfaces/clases correspondientes a las clases del metamodelo, mientras que las restantes están orientadas a dar soporte a la creación y procesamiento de tales modelos en forma activa.

4.3.3 Carga en memoria de un modelo persistido

Las herramientas que un entorno *MDDE* ofrece a los diseñadores de sistemas se encuentran incorporadas al entorno en forma de modelos conformes al metamodelo MDDE y almacenados persistentemente en una ubicación conocida. Para hacer uso de tales herramientas (lanzamiento de su ejecución y gestión de su flujo o su simple inspección), es necesario un procesamiento programático (interpretación) de dichos modelos, lo cual a su vez requiere poder cargarlos a memoria a partir de su forma persistida y así disponer de ellos en forma activa. El empleo de Eclipse/EMF como plataforma cubre perfectamente esta necesidad.

Soporte en Eclipse/EMF

El API de persistencia de EMF posibilita cargar fácilmente a memoria un modelo persistido en XMI. En consonancia con la subsección anterior, el resultado es una estructura de objetos donde cada uno es instancia de la correspondiente clase Java en la implementación que EMF permite obtener a partir del metamodelo cuyos modelos terminales se desea cargar.

El Código 4.4 a continuación muestra cómo se realiza la carga de un modelo representativo de una herramienta *MDDE* que se halla persistido en XMI como fichero `*.tool.mdde`.

```
File toolModelFile = ... ;
URI toolModelFileURI = URI.createFileURI(toolModelFile.getPath());

/*
 * I instantiate a ResourceSet for explicit resource creation and
 * register an XMI resource factory against the "mdde" file extension.
 */
ResourceSet resourceSet = new ResourceSetImpl();
resourceSet.getResourceFactoryRegistry().
            getExtensionToFactoryMap().put("mdde", new XMIResourceFactoryImpl());
XMIResource resource = (XMIResource)resourceSet.createResource(toolModelFileURI);

/* I load the model contents into the resource */
Map<String, Object> loadOptions = new HashMap<String, Object>(0);
loadOptions.put(... , ...);
...
resource.load(loadOptions);

/* I access to the model's main container */
Tool tool = (Tool)resource.getContents().get(0);
```

Código 4.4 – Carga de un modelo de herramienta

4.3.4 Ejecución de transformaciones M2M.

Dado que las transformaciones M2M son la operación fundamental en MDSE, en general las herramientas *MDDE* harán un uso extensivo de tareas basadas en M2M, cuya ejecución supone ejecutar la transformación en cuestión. Por ello, es esencial que la plataforma sobre la que implementar un entorno *MDDE* ofrezca la capacidad de ejecución de transformaciones M2M. El empleo conjunto de ATL y de Eclipse/EMF permite implementar esta funcionalidad.

Soporte en Eclipse/EMF

El componente ATL no sólo incluye un SDK para la codificación de transformaciones mediante ATL, sino que también proporciona la capacidad de ser éstas ejecutadas gracias a la máquina virtual de ATL[62].

4.3.5 Lanzamiento programático de la ejecución de transformaciones M2M

Consecuencia de la subsección anterior es requerir que la plataforma ofrezca la capacidad de lanzamiento programático de la ejecución de transformaciones M2M. De nuevo, el empleo conjunto de ATL y de Eclipse/EMF permite implementar esta funcionalidad.

Soporte en Eclipse/EMF

El API del componente ATL complementa al API de EMF y posibilita codificar el lanzamiento de la ejecución de transformaciones M2M implementadas mediante este lenguaje. Es importante notar que, aunque también es necesario manejar representaciones en memoria de los modelos y metamodelos involucrados, en este caso su obtención no se realiza en la forma convencional (uso del API de persistencia de EMF), sino con la alternativa ofrecida por el API de ATL.

El Código 4.5 a continuación muestra la implementación de esta acción para el caso más sencillo: transformación M2M 1:1.

```
String transfoPath = ... ;
String sourceMMPath = ... ;
String targetMMPath = ... ;
String inputModelPath = ... ;
String outputModelPath = ... ;
String sourceMMName = ... ;
String targetMMName = ... ;

/* Infrastructure Initialization */

CoreService.registerFactory("EMF model factory", EMFModelFactory.class);
EMFModelFactory factory = (EMFModelFactory)CoreService.getModelFactory("EMF model factory");
CoreService.registerInjector("EMF injector", EMFInjector.class);
EMFInjector injector = (EMFInjector)CoreService.getInjector("EMF injector");
CoreService.registerExtractor("EMF extractor", EMFExtractor.class);
EMFExtractor extractor = (EMFExtractor)CoreService.getExtractor("EMF extractor");
CoreService.registerLauncher("EMF-specific VM", EMFVMLauncher.class);
EMFVMLauncher launcher = (EMFVMLauncher)CoreService.getLauncher("EMF-specific VM");

/* Injection of involved artefacts */

EMFReferenceModel sourceMM = (EMFReferenceModel)factory.newReferenceModel();
Map<String, Object> injectionOptionsSourceMM = new HashMap<String, Object>(0);
injectionOptionsSourceMM.put(...,...);
...
injector.inject(sourceMM, new FileInputStream(sourceMMPath), injectionOptionsSourceMM);

EMFModel inModel = (EMFModel)factory.newModel(sourceMM);
Map<String, Object> injectionOptionsInputModel = new HashMap<String, Object>(0);
injectionOptionsInputModel.put(...,...);
...
injector.inject(inModel, new FileInputStream(inputModelPath), injectionOptionsInputModel);

EMFReferenceModel targetMM;
if(!targetMMName.equals(sourceMMName)) { // not endogenous transfo.
   targetMM = (EMFReferenceModel)factory.newReferenceModel();
   Map<String, Object> injectionOptionsTargetMM = new HashMap<String, Object>(0);
   injectionOptionsTargetMM.put(...,...);
   ...
```

[62] https://wiki.eclipse.org/ATL/Developer_Guide#ATL_Virtual_Machine

```
   injector.inject(targetMM, new FileInputStream(targetMMPath), injectionOptionsTargetMM);
} else
   targetMM = sourceMM;
EMFModel outModel = (EMFModel)factory.newModel(targetMM);

/* Launcher configuration and transformation launching */
Map<String, Object> initParams = new HashMap<String, Object>(0);
initParams.put(...,...);
...
launcher.initialize(initParams);
launcher.addInModel(inModel, "INmodel", sourceMMName);
launcher.addOutModel(outModel, "OUTmodel", targetMMName);
Object module = launcher.loadModule(new FileInputStream(transfoPath);
Map<String, Object> launchOptions = new HashMap<String, Object>(0);
launchOptions.put(...,...);
...
launcher.launch(ILauncher.RUN_MODE, new NullProgressMonitor(), launchOptions, module);

/* Persistence of created model */
FileOutputStream outModelFOS = new FileOutputStream(outputModelPath);
Map<String, Object> extractionOptions = new HashMap<String, Object>(0);
extractionOptions.put(...,...);
...
extractor.extract(outModel, outModelFOS, extractionOptions);
outModelFOS.close();
```

Código 4.5 – Lanzamiento programático de transformación M2M

4.3.6 Inicialización de nuevos modelos (conformes a un cierto metamodelo de dominio)

Cualquier usuario de un entorno *MDDE* necesitará a menudo crear nuevos modelos. Si se trata de un diseñador de sistemas que emplea un cierto entorno *MDDE* orientado a un determinado dominio, deseará crear modelos conformes a alguno de los metamodelos que formalizan el ámbito conceptual del entorno, mientras que si se trata de un diseñador de entornos específicos, su principal labor consistirá en formular modelos de nuevos tipos de tareas y de nuevas herramientas.

Por tanto, en *MDDE* se define el concepto de *inicializador de modelos*. Los elementos de este tipo tienen el propósito de facilitar al usuario la creación de nuevos modelos, adecuadamente inicializados según el caso. Para una implementación integrada y estandarizada de tales asistentes es muy conveniente que la plataforma de soporte escogida como base proporcione un mecanismo a partir del cual se pueda dotar de esta funcionalidad a un entorno *MDDE* a muy bajo coste. El empleo de Eclipse/EMF como plataforma de soporte posibilita cubrir esta funcionalidad.

Soporte en Eclipse/EMF

En Eclipse, su punto de extensión *org.eclipse.ui.wizards* permite definir declarativamente extensiones del *workbench* que representan asistentes (*wizards*) orientados a crear nuevos artefactos, importación, etc. Éstos resultan plenamente familiares y reconocibles como tal.

En el caso más sencillo (*wizard* de una sola página), la codificación de la funcionalidad propiamente dicha se realiza en simplemente un par de clases que, por ejemplo, pueden llamarse `MetamodelNameWizard` y `MetamodelNameWizardPage`. En la implementación del método `MetamodelNameWizardPage.createControl()` es donde se define la interfaz gráfica

de usuario del asistente. Una instancia de esta segunda clase se asocia a una instancia de la primera.

El punto donde codificar la funcionalidad del *wizard* es en la clase `MetamodelNameWizard`, implementando el método `performFinish()`. Por ejemplo, una posibilidad trivial (Código 4.6) es que el método cree un modelo activo, inicializado en cierta medida, lo salve a una ubicación según los datos introducidos por el usuario y finalmente abra el fichero recién almacenado con un editor adecuado para que el usuario complete la construcción del modelo.

```
public boolean performFinish() {

   /* Programmatic creation of a model, properly initialized */
   ...
   ...

   /* I save the programmatically created model to a file */
   ...
   ...

   /* I open the persisted file in an editor */
   ...
   ...

   return true;
}
```

Código 4.6 – Ejemplo de funcionalidad de wizard

El ejemplo anterior permite identificar las siguientes dos funcionalidades básicas que han de venir soportadas por la plataforma sobre la que se construyan entornos *MDDE*.

- Construcción programática de un modelo (activo).
- Almacenamiento persistente de un modelo activo.

4.3.7 Construcción programática de un modelo (activo)

Soporte en Eclipse

El empleo de Eclipse/EMF posibilita que la construcción programática de modelos conformes a un metamodelo sea sencilla, utilizando la implementación Java de éste producida por EMF. El Código 4.7 a continuación lo ejemplifica en base a un supuesto metamodelo llamado *Domain* cuya clase contenedor principal se llama `DomainModel`. Ésta define la asociación-composición `elements` mediante la que el objeto contenedor principal de cualquier modelo *Domain* contiene al resto de elementos del modelo.

```
DomainFactory factory = DomainFactory.eINSTANCE;
DomainModel mainContainer = factory.createDomainModel();
mainContainer.setAttr1(...);
mainContainer.setAttr2(...);
...
Class1 obj1 = factory.createClass1();
obj1.set..;
...
Class2 obj2 = factory.createClass2();
obj2.set..;
...
mainContainer.getElements().add(obj1);
mainContainer.getElements().add(obj2);
...
```

Código 4.7 – Ejemplo de construcción programática de modelos

4.3.8 Almacenamiento persistente de un modelo activo

Soporte en Eclipse

Al igual que en la carga de modelos a memoria, el empleo de Eclipse/EMF como plataforma de soporte cubre perfectamente esta funcionalidad, pues el API de persistencia de EMF posibilita salvar fácilmente un modelo desde memoria. El Código 4.8 a continuación muestra cómo se realiza la persistencia del modelo construido en el apartado anterior. El resultado es un fichero XMI con extensión "dom".

```
String modelFilePath = ... ;
File modelFile = new File(modelFilePath);
URI modelFileURI = URI.createFileURI(modelFile.getPath());
/*
 * I instantiate a ResourceSet and
 * register an XMI resource factory (EMF's default factory for resources)
 * against the ".dom" file extension
 */
ResourceSet resourceSet = new ResourceSetImpl();
resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put("dom", new
                                                                    XMIResourceFactoryImpl());
// I create an empty resource explicitly
XMIResource resource = (XMIResource)resourceSet.createResource(modelFileURI);
// I add the main container to the resource and save it
resource.getContents().add(mainContainer);
Map<String, Object> saveptions = new HashMap<String, Object>(0);
saveOptions.put(...,...);
...
resource.save(saveOptions);
```

Código 4.8 – Persistencia de modelo existente en memoria

4.4 Ejemplo de entorno MDDE: *MDDE-MinimalMAST2*

En esta sección se describen los elementos básicos de un entorno *MDDE* muy simple, diseñado a modo de prueba de concepto y denominado *MDDE-MinimalMAST2*. Se trata de un entorno orientado al diseño y análisis de SDTREs mediante las herramientas legadas de MAST-1.x.

> ☞ La descripción completa y detallada del entorno *MDDE-MinimalMAST2* puede encontrarse en http://www.istr.unican.es/members/cesarcuevas/phd/.html

4.4.1 Herramientas

En el diseño del entorno *MDDE-MinimalMAST2* se han seleccionado tres herramientas, cuyas invocaciones son diferentes:

- **MAST2_Simulation**. Permite obtener el comportamiento temporal de peor caso y de caso promedio de un SDTRE utilizando la herramienta de simulación *JSimMAST2*.

  JSimMAST2 es una herramienta legada desarrollada en Java y que, por tanto, puede ser ejecutada en la propia plataforma de soporte donde se ejecuta el entorno.

- **MAST2_Analysis**. Permite realizar el análisis de planificabilidad de un SDTRE cuyo comportamiento temporal esté descrito mediante un modelo MAST-1.4, utilizando cualquiera de las herramientas de análisis disponibles en MAST. En este caso las

herramientas que realizan el análisis también son legadas, pero están implementadas en Ada y, por tanto, han de ser ejecutadas en una plataforma de ejecución diferente a la del entorno.

- **Create_MAST2model**. Posibilita la creación de un modelo MAST-2 representativo del comportamiento temporal de un SDTRE utilizando el editor estándar en árbol que proporciona Eclipse.

4.4.2 Tipos de tareas

La funcionalidad de las herramientas anteriores se implementa haciendo uso de los siguientes tipos de tareas. Por consiguiente, éstos tienen que estar disponibles en el entorno *MDDE-MinimalMAST2*.

- **ConstraintsVerification**. Consiste en chequear el cumplimiento por parte de un modelo de las restricciones especificadas sobre su metamodelo, tanto inherentes como específicas de herramientas, generando como resultado un modelo de diagnóstico.
- **MAST2_to_JSimMAST2**. Consiste en generar el modelo de simulación JSimMAST2 correspondiente a un modelo MAST-2, de acuerdo con el perfil de simulación que se establezca.
- **Exec_JSimMAST**. Tarea adaptadora de artefacto que ejecuta el simulador JSimMAST2, utilizando la zona de área gráfica para mostrar la interfaz gráfica del simulador.
- **Exec_MAST**. Tarea adaptadora de artefacto externo que se comunica y controla la herramienta externa legada MAST. La interfaz de usuario de la herramienta se representa en una ventana auxiliar que se superpone a la ventana del entorno.
- **Finish_Tool**. Se reduce a la finalización de la ejecución de cualquier herramienta, cerrando su cadena de tareas.

4.4.3 Herramientas (Procesos)

MAST2_Simulation

Esta herramienta permite analizar mediante simulación el comportamiento temporal de SDTREs modelados mediante MAST-2. El elemento central de la herramienta es el simulador JSimMAST2. De forma previa a ejecutar la simulación, la herramienta verifica en el modelo de entrada el cumplimiento de las restricciones de integridad especificadas sobre el metamodelo de MAST-2, así como de las restricciones específicas de JSimMAST2. También es necesario seleccionar un perfil para la ejecución de la simulación y, en base a él, obtener el modelo de simulación (conforme al metamodelo SimMAST2) a partir del modelo MAST-2 de entrada.

Esta herramienta actúa sobre un modelo de entrada MAST-2 y produce como salida un modelo MAST-2 de Resultados y, opcionalmente, un modelo MAST-2 de Trazas.

Información de configuración

La información de configuración que requiere la herramienta es:

- Obligatoria:
  - Modo de ejecución (Continuo / Paso a paso).
  - Modelo MAST-2 a simular.

- Opcional:
  - --

Tareas constituyentes

La Figura 4.20 muestra las tareas que constituyen la herramienta:

T-S1. **Verificación del modelo MAST-2** de entrada respecto de las *Restricciones MAST-2*.

T-S2. **Verificación de compatibilidad con JSimMAST2**, esto es, verificación de que el modelo MAST-2 de entrada cumple las restricciones específicas de JSimMAST2.

T-S3. **Generación de modelo simulable por JSimMAST2**, esto es, transformación del modelo MAST-2 de entrada en un modelo conforme al metamodelo SimMAST2, en función del perfil de simulación deseado.

T-S4. **Ejecución de JSimMAST2** sobre el modelo simulable, según el mismo perfil de simulación establecido en la tarea anterior.

T-S5. **Conclusión del proceso**.

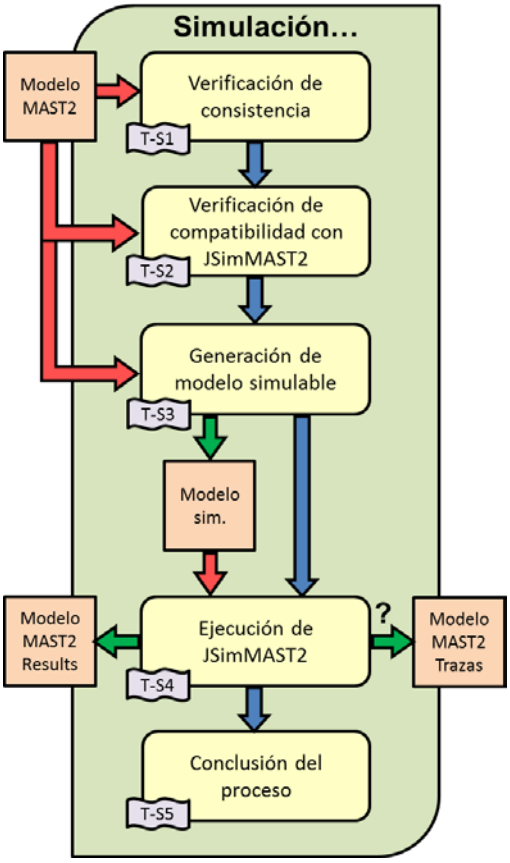


Figura 4.20 – Tareas constituyentes de la herramienta MAST2_Simulation.

Las tareas expuestas se encuentran ligadas entre si a través de los modelos generados como salida y aquellos requeridos como entrada, así como mediante el perfil de simulación escogido:

- El modelo de entrada a la T-S1 es también modelo de entrada a las T-S2 y T-S3.
- El modelo de salida de la T-S3 es modelo de entrada para la T-S4.

- El perfil de simulación establecido para la T-S3 ha de coincidir con el perfil de simulación establecido para la T-S4, pues según cuál vaya a ser ese perfil, la generación del modelo simulable sigue unas directrices u otras.

Información de estatus

Durante su ejecución, la herramienta muestra la siguiente información de estatus:

- Tarea en curso.
- Estado de la ejecución del proceso.

Herramienta MAST2_Analysis

Esta herramienta permite realizar diversos tipos de análisis de planificabilidad sobre SDTREs modelados mediante MAST-2, empleando las herramientas de análisis existentes para MAST-1.4. Previamente a ejecutar el análisis propiamente dicho, el proceso contempla la verificación de que el modelo cumple tanto las restricciones de integridad especificadas sobre el metamodelo de MAST-2 como las restricciones de compatibilidad con MAST-1.4 y las específicas de la técnica de análisis escogida. Satisfechas las verificaciones previas, se invoca la herramienta de análisis, y a través de la ventana de interacción que ofrece se selecciona el tipo de análisis a ser realizado.

Esta herramienta actúa sobre un modelo de entrada MAST-2 y produce como salida un modelo MAST-2 de Resultados y, opcionalmente, un modelo MAST-2 con parámetros de planificación calculados por la herramienta.

Información de configuración

La información de configuración que requiere la herramienta es:

- Obligatoria:
  - Modo de ejecución (Continuo / Paso a paso).
  - Modelo MAST-2 a analizar.
- Opcional:
  - --

Tareas constituyentes

La Figura 4.21 muestra las tareas que constituyen la herramienta:

T-A1 **Verificación del modelo MAST-2** de entrada respecto de las *Restricciones MAST-2*.

T-A2 **Verificación de compatibilidad respecto a MAST-1.4**, esto es, verificación de que el modelo MAST-2 de entrada cumple las restricciones de compatibilidad con MAST-1.4.

T-A3 **Verificación de compatibilidad respecto a la técnica de análisis** escogida, esto es, verificación de que el modelo MAST-2 de entrada cumple las restricciones propias de la técnica de análisis.

T-A4 **Ejecución del análisis de planificabilidad** sobre el modelo MAST-2 de entrada, según la misma técnica de análisis de planificabilidad establecida en la tarea anterior.

T-A5 **Conclusión del proceso**.

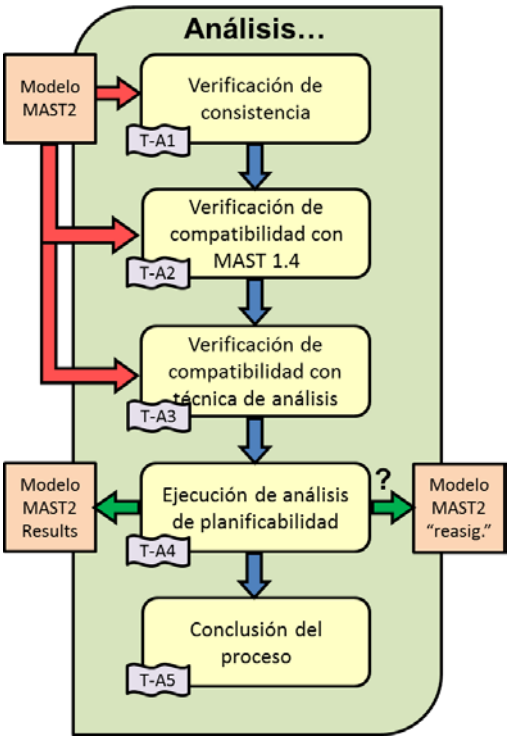


Figura 4.21 – Tareas constituyentes de la herramienta MAST2_Analysis.

Las tareas expuestas se encuentran ligadas entre si a través de los modelos generados como salida y aquellos requeridos como entrada, así como mediante la técnica de análisis escogida:

- El modelo de entrada a la T-A1 es también modelo de entrada a las T-A2 y T-A3.
- La técnica de análisis establecida para la T-A3 ha de coincidir con la técnica establecida para la T-A4.

Información de estatus

En cualquier momento durante su ejecución, el proceso mantiene y expone la siguiente información de estatus:

- Tarea en curso.
- Estado de la ejecución del proceso.

Create_MAST2Model

Esta herramienta facilita al operador la creación del modelo MAST-2 de un SDTRE utilizando el editor en árbol que proporciona Eclipse, en base a que conoce el metamodelo MAST-2. El proceso contempla comenzar la construcción del modelo bien desde cero o bien a partir de un contenido inicial (un submodelo de plataforma y/o uno lógico y/o uno reactivo ya existentes) y no concluye una vez se da por finalizada la construcción, sino que también engloba la verificación tanto de que el modelo construido es estructuralmente correcto como de que satisface las *Restricciones MAST-2*.

Esta herramienta puede actuar sobre uno o más modelos de entrada MAST-2 y produce como salida otro modelo MAST-2, estructuralmente bien formado y que cumple todas las restricciones de integridad MAST-2.

Información de configuración

La información de configuración que requiere esta herramienta es:

- Obligatoria:
  - Modo de ejecución (Continuo / Paso a paso).
  - Localización donde crear el modelo MAST-2 (ruta + nombre).
- Opcional:
  - Localizador del modelo con el contenido inicial.

Tareas constituyentes

La constitución de la herramienta está basada en las siguientes tareas:

T-C1. **Inicialización** del modelo MAST-2.

T-C2. **Construcción** del modelo MAST-2.

T-C3. **Verificación** de que el modelo MAST-2 construido es estructuralmente correcto (conforme) y de que satisface las *Restricciones MAST-2*.

T-C4. **Conclusión** del proceso.

Estas tareas se encuentran ligadas entre sí a través de los modelos generados como salida y aquellos requeridos como entrada:

- El modelo de salida de la T-C1 es modelo de entrada a la T-C2.
- El modelo de salida de la T-C2 es modelo de entrada a la T-C3.

Información de estatus

En cualquier momento durante su ejecución, el proceso mantiene y expone la siguiente información de estatus:

- Tarea en curso.
- Estado de la ejecución del proceso.

5 Conclusiones y futuras líneas de trabajo

5.1 Conclusiones

Se procede a resumir las contribuciones realizadas en el marco de esta Tesis en respuesta a los objetivos marcados en su inicio y presentados en el capítulo 1 de esta memoria. En él, se ha defendido y contrastado que la aplicación de la disciplina MDSE es actualmente la estrategia más adecuada para dar soporte a los entornos SDTRE. Partiendo de esta premisa, el objetivo central de la Tesis ha sido proporcionar nuevas estrategias y nuevos recursos que faciliten el uso de MDSE y sus tecnologías a los expertos en Ingeniería Software que diseñan tales entornos.

En particular, se ha abordado el objetivo específico de que estos expertos puedan hacer sus contribuciones para evolucionar los entornos a través de la formulación de modelos relacionados con los dominios en que son expertos, y no a través del desarrollo o modificación de código basado en el conocimiento y uso de la infraestructura MDSE que en gran medida es ajena a ellos.

Los siguientes apartados identifican y describen las contribuciones específicas que se han aportado en esta Tesis.

Identificación de los agentes relacionados con los entornos SDTRE

El punto de partida de la Tesis ha sido la caracterización y asignación de responsabilidades a los tres tipos de agentes que tienen relación (hacen uso y/o contribuyen) con los entornos SDTRE:

- El ***desarrollador de sistemas software*** es el destinatario final del entorno, que lo utiliza para desarrollar los sistemas y aplicaciones específicas que demandan la industria o la sociedad. Para él, el entorno de desarrollo es la herramienta y plataforma donde lleva a cabo su trabajo.
- El ***diseñador de entornos de desarrollo*** es el ingeniero software experto en dominios y tecnologías de software de SDTREs (especificación, arquitectura, diseño, análisis de planificabilidad, fiabilidad, verificación y pruebas, etc.) y sus responsabilidades consisten en diseñar el entorno, proponer procesos de desarrollo y dotarlo de modelos de información y herramientas de desarrollo apropiadas.
- Por último, el ***desarrollador de infraestructuras de entorno*** es el experto en tecnologías MDSE que tiene como responsabilidades proporcionar el ámbito y la tecnología de soporte del entorno así como generar los elementos de abstracción y adaptación necesarios para que los entornos permanezcan y se puedan mantener aun cuando evolucione la tecnología MDSE de soporte.

La contribución de la Tesis es el reconocimiento y la diferenciación clara entre los dos últimos agentes, y la concepción y prueba de concepto de cómo puede liberarse al *diseñador de entornos de desarrollo* para que pueda ser sólo experto en su dominio específico y no tenga que ser también experto en tecnologías MDSE.

Vertebración de los entornos SDTRE en base a la integración de diferentes paradigmas de representación de la información (espacios tecnológicos)

En la Tesis se ha adoptado el concepto de *Espacio Tecnológico* (TS) para denotar desde un alto nivel de abstracción tanto una forma específica y diferenciada de representar la información del sistema que se desarrolla como sus tecnologías asociadas. Se han identificado y caracterizado

los espacios que constituyen la base de los entornos SDTRE y se defiende que el núcleo de estos entornos es la infraestructura que integra estos espacios tecnológicos, de forma que la información del sistema bajo desarrollo, aun siendo única para garantizar su coherencia, pueda ser accedida por cualquiera de los espacios según su destino.

Se han analizado tres espacios tecnológicos esenciales.

- ***Modelware***. La información de los modelos se representa de acuerdo a una arquitectura de tres capas que se deriva de metametamodelos tales como Ecore o EMOF. Este TS proporciona un modelo de referencia para la representación de la información en la memoria de las aplicaciones y herramientas, de forma que sea accesible desde el código de forma estandarizada, natural y eficiente.
- ***Grammarware***. La información se representa mediante DSLs textuales. Este TS ofrece la información al experto de dominio de forma natural, amigable y libre de los conceptos de la tecnología MDSE de soporte.
- ***XMLware***. La información se formula como secuencias de caracteres textuales estándares compatibles con cualquier plataforma. Este TS proporciona la información con formatos adecuados para que pueda ser almacenada en repositorios persistentes y para que pueda ser intercambiada con otros entornos que utilizan otras tecnologías diferentes.

Aunque no tratado en la Tesis, se contempla la incorporación de otros espacios tecnológicos que, por ejemplo, representen la información en base a ontologías de forma que se pueda gestionar mediante técnicas basadas en inteligencia artificial o que incorporen la representación de la información en formatos compatibles con diferentes tipos de BBDD, a fin de asegurar la escalabilidad del entorno.

Análisis y justificación de la formulación laxa de metamodelos

Se ha abordado el problema de la determinación del nivel de detalle que conviene alcanzar al formalizar un dominio mediante metamodelado. Tras analizar las desventajas, incluso imposibilidad en muchos casos, de realizar una formulación de completa rigurosidad, se propone adoptar y asumir que las formulaciones de los metamodelos van a ser laxas.

Sin embargo, el uso de metamodelos laxos requiere que el entorno proporcione recursos en su tecnología de base que permitan de forma automática y genérica delimitar la laxitud de los metamodelos, de forma que cuando un modelo sea suministrado con un objetivo (para su procesamiento con una herramienta, para su interpretación por un diseñador, etc.) se tenga la seguridad de que la información del modelo es válida para él, y no necesite tener que ser verificada su coherencia antes de su uso. Para ello se prevé que todo metamodelo esté complementado por diferentes conjuntos de restricciones, relacionadas en unos casos con la propia laxitud de los metamodelos y en otros con las herramientas destinatarias de los modelos. Bajo este prisma, la coherencia de los modelos respecto al fin a que se destinan ha de ser verificada por herramientas que contrastan la información con los correspondientes conjuntos de restricciones.

Esta estrategia se ha ejemplificado en base al metamodelo MAST-2. Su formulación incluye numerosas laxitudes y va a ser procesada por muy diferentes herramientas, por lo que ha constituido un banco de pruebas ilustrativo de la estrategia propuesta.

Análisis de la utilización de DSMLs textuales para presentar la información y asistencia a su generación automática en base a metamodelos

Se ha realizado un estudio de las ventajas de adoptar el enfoque específico de dominio, en particular en su vertiente textual, como forma de acercar la información a los expertos de dominio que la han de interpretar. Una vez establecida su idoneidad, se ha abordado la visión más comúnmente adoptada actualmente para la formalización de los DSMLs y se ha llevado a cabo una revisión de las plataformas existentes para la generación automática de lenguajes textuales. El trabajo se ha centrado en la herramienta Xtext proporcionada en la plataforma Eclipse.

Como prueba de aplicación, se ha comparado el lenguaje que genera Xtext para el metamodelo MAST-2 y el que ya existía nativo hecho por expertos. La generación del lenguaje con la herramienta Xtext configurada tal y como es suministrada en Eclipse es inmediata. Sin embargo, el lenguaje generado resulta algo redundante respecto a lo que espera el experto. Un experto en MDSE puede modificar Xtext para que genere el lenguaje que se necesita, pero difícilmente esta tarea puede ser asumida por los expertos de dominio. Por ello, la contribución de la Tesis ha sido proporcionar procedimientos y herramientas que a través de modelos que son fácilmente asumidos por los expertos de dominio, permitan configurar Xtext para que genere los lenguajes de dominio que se necesiten. A la exposición de las líneas más significativas de tal desarrollo se ha dedicado un apartado completo de esta memoria.

Definición de un modelo organizativo para la información persistida en un entorno

Puesto que la información involucrada en un proceso de desarrollo de SDTREs es muy compleja, polifacética y altamente interconectada, la disciplina MDSE propone la formulación de la información global de un entorno SDTRE segmentada en conjuntos de múltiples modelos interreferenciados, permitiendo así compatibilizar la separación de los diferentes aspectos con la coherencia del conjunto.

La persistencia de tales modelos en los entornos implica la conveniencia de un modelo organizativo que permita su explotación ágil y eficiente. Por ello, en la Tesis se ha propuesto un modelo organizativo que ha resultado del análisis de diversos criterios para la clasificación de los modelos, de soluciones de localización de la información y de estrategias básicas de organización de los modelos. Como prueba de concepto se ha formulado una organización específica para ser implementada en la plataforma Eclipse haciendo uso de su paradigma de gestión de recursos.

Estrategias de diseño de herramientas MDSE genéricas

La aplicación de la disciplina MDSE como base de los entornos presenta el hándicap de la dependencia entre el código de las herramientas y el contenido de los metamodelos constitutivos del ámbito conceptual de los mismos, hándicap que se acentúa en la medida en que éstos evolucionen frecuentemente (como es el caso de los entornos SDTRE). Como solución global se propone fomentar el uso de herramientas genéricas en los entornos, esto es, herramientas aplicables a modelos conformes a diferentes metamodelos. Para ello, su funcionalidad ha de venir descrita mediante lo que se ha denominado un *espacio conceptual común* (ECC), de forma que su código es válido siempre que se mantenga inalterado el ECC.

Se han identificado algunos casos sencillos de ECC, como el metametamodelo y la herencia o extensión a nivel de metamodelos, para los que las herramientas MDSE desarrolladas según las estrategias habituales resultan genéricas. Sin embargo, la principal contribución de la Tesis en este aspecto es la propuesta de una estrategia mucho más general para construir herramientas genéricas: herramientas basadas en un metamodelo de instrucción donde cada modelo (instructor) conforme a él adapta (instruye) la herramienta a cada caso. Más concretamente, se ha apostado por metaherramientas, las cuales consiguen la adaptación gracias a que generan bajo demanda la herramienta específica apropiada.

La estrategia se ha validado mediante el diseño de tres metaherramientas.

Metaherramienta para verificación del cumplimiento de restricciones especificadas sobre un metamodelo

Como punto de partida, se ha propuesto considerar la verificación de modelos como una transformación M2M que, a partir del modelo a verificar, genere un modelo de diagnóstico que describa los resultados de la verificación. En base a esta visión, se ha diseñado una herramienta genérica basada en metamodelo de instrucción en forma de metaherramienta (basada en una HOT) que genera bajo demanda la herramienta específica (transformación) a ser aplicada en función del metamodelo de dominio y las restricciones especificadas sobre él.

Se ha definido y formulado mediante Ecore su correspondiente metamodelo de instrucción, llamado *Constraints Characterization* (`CC.ecore`), cuyos modelos terminales constituyen la entrada de la metaherramienta. Cada modelo representa un conjunto de restricciones adecuadamente caracterizado. Además, puesto que la estrategia básica para verificación se basa en aplicar una transformación M2M de chequeo, se ha desarrollado un segundo metamodelo, llamado *Constraints Violation Description* (`CVD.ecore`), que formaliza la estructura de los modelos de diagnóstico, resultado de la verificación. Estos modelos incluyen la formulación de los incumplimientos existentes en los modelos chequeados. Asimismo, también se ha expuesto la estructura uniforme de las transformaciones de chequeo generadas por la metaherramienta y se ha desarrollado la especificación de la HOT y su codificación en ATL (`CC_to_ATL.atl`). Por razones de espacio y excesiva especificidad para una memoria, esta información detallada sobre la HOT no se incluye, aunque se ofrecen punteros a la localización web donde puede consultarse.

Como ejemplo de aplicación, se ha expuesto la actuación de la metaherramienta en el ámbito MAST-2. Puesto que para el metamodelo MAST-2, de formulación laxa, se encuentra definido un conjunto de restricciones de integridad, la caracterización de tal conjunto ha sido formulada como modelo conforme al metamodelo CC (`Mast2_integrity.cc.xmi`). El modelo construido ha servido de entrada a la metaherramienta que, mediante su HOT nuclear, ha producido la correspondiente transformación de chequeo (`Mast2_integrity_to_CVD.atl`). Posteriormente, se han realizado similares pruebas a partir de otros conjuntos de restricciones sobre el metamodelo MAST-2, en este caso específicas de herramienta.

Metaherramienta para construcción de modelos acordes a vistas restrictivas

Se ha presentado el problema de la construcción de modelos que, además de ser conformes al metamodelo, cumplan una serie de condicionantes o restricciones estructurales recogidas bajo el concepto de *vista* especificada sobre el metamodelo. Se han analizado opciones para llevar a cabo la construcción de modelos acordes a una vista dada y se ha propuesto una metodología

consistente en disponer de un metamodelo que describa los datos requeridos por la vista, llamado en general metamodelo *View Required Data* (VRD) y que gobierne el proceso de construcción de modelos, de manera que el diseñador sólo afronte la complejidad de la vista. Posteriormente una transformación M2M produce el modelo final conforme al metamodelo original y acorde a la vista.

El alcance del concepto de vista ha sido establecido y para la automatización de la estrategia se ha propuesto una herramienta genérica basada en metamodelo de instrucción en forma de metaherramienta (basada en una pareja de transformaciones M2MM y HOT). Se ha diseñado y formulado mediante Ecore el metamodelo de instrucción, llamado *Constraining View Specification* (`CVS.ecore`). A partir de un modelo conforme a CVS que formula la descripción de una vista, la metaherramienta opera en dos pasos, generando sucesivamente el metamodelo VRD que da soporte a la construcción de modelos por parte del diseñador y la transformación M2M que produce el modelo final, denotada en general como *VRD to Domain*. Se han desarrollado las especificaciones de las correspondientes transformaciones (promocionadora y HOT) que implementan la metaherramienta y su codificación en ATL (`CVS_to_Ecore.atl` y `CVS_to_ATL.atl`). De nuevo por razones de espacio y excesiva especificidad para una memoria, esta información detallada no se incluye, aunque se ofrecen punteros a la localización web donde se encuentra alojada.

Como ejemplo de aplicación, se ha expuesto la actuación de la metaherramienta en el ámbito MAST-2. Se ha propuesto un escenario en el que contextualizar una vista restrictiva sobre el metamodelo MAST-2, llamada *LinuxClassicRMA*, la cual ha sido formulada como modelo CVS (`LCRMA.cvs.xmi`). El modelo construido ha servido de entrada a la metaherramienta que, mediante sus transformaciones nucleares, ha producido el correspondiente metamodelo VRD (`VRD_for_Mast2-LCRMA.ecore`) y la correspondiente transformación final (`VRD_for_Mast2-LCRMA_to_Mast2.atl`).

Metaherramienta de interoperabilidad XML ↔ Modelware

Se ha abordado el problema de trabajar en un dominio formalizado mediante un *W3C-Schema* y un metamodelo que han sido desarrollados independientemente. En primer lugar, se ha presentado una estrategia para la conversión entre los correspondientes documentos XML y modelos, basada en la concatenación de proyecciones entre los dos espacios tecnológicos involucrados y transformaciones M2M. Éstas últimas son transformaciones *ad hoc*, pues son dependientes de las formalizaciones a nivel de W3C-S*chema* de las diversas primitivas de metamodelado. Aunque la variedad de posibles formalizaciones es prácticamente ilimitada, se ha contemplado un conjunto representativo de casos y se ha diseñado una herramienta genérica para interoperabilidad XML ↔ *Modelware* con una aplicabilidad razonablemente general. De nuevo, es una herramienta genérica basada en metamodelo de instrucción en forma de metaherramienta (basada en dos HOT).

En este caso no se ha desarrollado un metamodelo de instrucción, sino que se ha reutilizado el metamodelo `Mapping.ecore` proporcionado por EMF para formulación de modelos de correspondencias (*mappings*) totalmente genéricas. Además, puesto que estos modelos de instrucción han de relacionar elementos de un metamodelo con elementos de un *schema*, es necesario disponer de éstos últimos en forma de modelo, para lo cual se ha hecho uso de otro metamodelo también proporcionado por EMF, el metamodelo `XSD.ecore`. Sí se han

desarrollado las especificaciones de las correspondientes HOT que implementan la metaherramienta (producen las transformaciones M2M de interoperabilidad en ambos sentidos) y su codificación en ATL (`Mapping_to_ATL_for_XML_to_Modelware.atl` y `Mapping_to_ATL_for_Modelware_to_XML.atl`). De nuevo por razones de espacio y excesiva especificidad para una memoria, esta información detallada no se incluye, aunque se ofrecen punteros a la localización web donde se encuentra alojada.

Como ejemplo de aplicación, se ha expuesto la actuación de la metaherramienta en el ámbito MAST-2, dominio en el que se cumple la premisa inicial de formalización mediante *W3C-Schema* y metamodelo desarrollados independientemente. De hecho, a partir de este escenario ha sido posible identificar buena parte de las estrategias de formalización a nivel de *W3C-Schema* que contempla la metaherramienta. Se ha obtenido la representación del *schema* `Mast2_Model.xsd` como modelo XSD (`Mast2_Model.xsd.xmi`) y se ha construido el modelo de *mappings* que lo relaciona con el metamodelo MAST-2 (`Mast2.mapping.xmi`). El modelo construido ha servido de entrada a la metaherramienta que, mediante sus dos HOT nucleares, ha producido las correspondientes transformaciones de interoperabilidad (`Modelware_to_XML_for_Mast2.atl y XML_to_Modelware_for_Mast2.atl`).

Modelo de referencia para el diseño de entornos de desarrollo en base a modelos

El diseño de un entorno SDTRE no sólo requiere diseñar metamodelos que formalicen el soporte de la información y herramientas que lleven a cabo las transformaciones de ésta, sino que también requiere diseñar procesos de desarrollo que engloban conjuntos de modelos generados encadenando y/o iterando la aplicación de herramientas existentes en el entorno bajo la supervisión del operador. Aunque la concepción de estos procesos de desarrollo sea responsabilidad del agente *diseñador de entornos*, que concibe las estrategias con las que se va a utilizar el entorno, su implementación en base a la infraestructura MDSE proporcionada por la plataforma que da soporte al entorno queda, por su complejidad, fuera de su capacidad y conocimiento.

Para facilitar la tarea de este agente, se ha propuesto una concepción genérica de entornos basados en MDSE, denominada MDDE (*Model-Driven Development Environment*). Ésta incluye la definición de un modelo de referencia para el diseño de entornos y de un conjunto de recursos de soporte que facilitan al experto diseñador de entornos su especificación e implementación. Siempre que el diseñador acepte el modelo de referencia, los procesos se definen en base a modelos que describen los modelos y herramientas que participan y las interacciones requeridas con el operador. Estos modelos descriptivos de procesos son interpretados por una herramienta ofrecida por el entorno, permitiendo su ejecución.

Definición del modelo de referencia desde un punto de vista estructural

Se han descrito los elementos conceptuales que constituyen el modelo de referencia, en base a sus atributos así como a sus interrelaciones, especificando para cada uno de ellos sus tipos derivados y su clasificación. En esencia, el modelo de referencia propuesto considera que el operador que utiliza un entorno lleva a cabo su actividad mediante la ejecución supervisada de procesos, la cual a su vez consiste en la ejecución secuencial o iterativa de operaciones más básicas denominadas tareas. De acuerdo con el espíritu de la disciplina MDSE, tanto los procesos como los tipos de tareas instanciados en ellos se formulan mediante modelos, de manera que

la especificación y diseño de un entorno consiste básicamente en la elaboración de modelos, y no en el desarrollo de su código de implementación. En este sentido, se ha definido el metamodelo que formaliza los modelos de procesos y tipos de tareas que constituyen cada entorno concreto. Asimismo, como prueba de concepto se ha diseñado e implementado una herramienta que los interpreta y genera automáticamente los recursos del entorno diseñado. Por último, se ha definido la GUI normalizada (*MDDE workbench*) que todo entorno acorde a esta concepción ha de presentar.

Análisis de aspectos a los que ha de dar soporte funcional la plataforma que sirve de base de entornos MDDE

Se ha analizado la funcionalidad básica que ha de proveer la plataforma que ha de dar soporte a un entorno definido en base al modelo de referencia MDDE, de forma que, entre otras cosas, permita la implementación a bajo coste del *MDDE workbench*. También se han analizado las funcionalidades a nivel de usuario final que cada entorno *MDDE* ha de implementar, y a partir de ellas se han identificado otras de más bajo nivel, relacionadas principalmente con el manejo de modelos, en base a las que se implementan las primeras. La plataforma base también es responsable del soporte a tales acciones básicas. Se ha justificado a través de pruebas de concepto la validez de Eclipse / EMF como plataforma de soporte.

Desarrollo de un prototipo de entorno MDDE.

La concepción *MDDE* formulada a través del modelo de referencia y la validez de la plataforma Eclipse/EMF como plataforma de soporte han sido validadas a nivel de prueba de concepto mediante la construcción de un entorno concreto, que se ha sido denominado *MDDE-MinimalMAST2*, orientado al desarrollo de SDTREs utilizando la metodología MAST.

5.2 Trabajo futuro

El trabajo desarrollado en esta Tesis plantea un conjunto diverso de líneas de trabajo que se pueden abordar a partir de él, algunas de las cuales ya se han iniciado.

Desarrollo de un entorno MDDE completo para desarrollo de SDTREs.

El prototipo de entorno desarrollado como prueba de concepto de las aportaciones de esta Tesis se centra en la fase de diseño y validación del comportamiento de tiempo real de un sistema con restricciones temporales haciendo uso de la herramienta MAST. Sin embargo, el máximo beneficio del uso de las estrategias propuestas (o de la estrategia MDDE en general) en el desarrollo de SDTREs se obtendrá cuando se aborde el ciclo de vida completo del sistema, utilizando un único entorno de desarrollo que dé soporte centralizado a todas las fases del mismo.

Típicamente, los esfuerzos de la comunidad investigadora en torno a la utilización de técnicas dirigidas por modelos en el desarrollo de SDTREs han estado centrados especialmente en las fases de diseño e implementación, dejando a un lado otras fases como la especificación de requisitos o la validación. Por tanto, una línea de investigación que parte de esta Tesis es el desarrollo de una metodología y su correspondiente IDE, plenamente dirigidos por modelos, que utilizando los técnicas aquí desarrolladas den soporte integrado a todas las fases del ciclo de vida completo de un sistema con requisitos temporales, desde las fases iniciales de especificación hasta la implementación y validación completa del sistema.

La metodología deberá establecer las fases, y actividades a realizar dentro de cada fase, a seguir para llevar a cabo la especificación y posterior refinamiento de los requisitos iniciales del sistema (con especial énfasis en los requisitos no funcionales), de manera que sean transformados a decisiones de diseño que puedan ser validadas y garantizadas a nivel de implementación. Desde el punto de vista del entorno, deberá constituir el modelo de referencia para la integración de las herramientas que procesan la información utilizada en las diferentes fases del proceso de desarrollo de SDTREs.

Esta extensión del alcance del entorno de desarrollo servirá para evaluar las aportaciones realizadas en esta Tesis, comprobando si son suficientes para dar soporte a los diferentes aspectos del ciclo de vida de un sistema, o si por el contrario, han de ser extendidas o adaptadas según los diferentes puntos de vista involucrados. Aunque es de esperar la aparición de nuevas necesidades respecto a metaherramientas, los fundamentos aquí definidos deberían ser suficientes para su desarrollo.

Escalabilidad completa del espacio de datos de las herramientas

Se ha observado que las herramientas desarrolladas con las tecnologías MDSE de Eclipse presentan problemas de escalabilidad cuando se necesita tener simultáneamente un gran número de modelos, representados en forma de objetos según su descripción EMF/Ecore, en la memoria primaria de las aplicaciones Java que las implementan. La solución disponible es almacenar y recuperar sucesivamente los modelos en memoria secundaria, y esto da lugar a tiempos de latencia muy altos.

En particular, si la herramienta hace uso de operaciones iterativas en las que en los sucesivos pasos se van ajustando o refinando los modelos, los tiempos de ejecución que se requieren hacen que las herramientas sean inutilizables. Una línea de trabajo que ya se ha iniciado es la utilización de una representación nativa de los modelos en una BBDD orientada a grafos que proporcione una escalabilidad completa al espacio de datos accesible desde el código de las herramientas y aplicaciones que se integren en el entorno. De esta forma, la herramienta construye una BBDD local en la que los elementos del modelo están representados de forma primaria por estructuras (grafos) de la BBDD y el código de la herramienta gestiona los modelos sin requerir que la información tenga que ser representada de otra forma alternativa (por ejemplo instancias de clases Java) en la memoria de ejecución de la herramienta. Este enfoque difiere en el objetivo del uso de las BBDD con los proyectos MDSE (como en el proyecto CDO de Eclipse) ya que no se trata de proporcionar persistencia escalable a los modelos de forma individual y proporcionar garantía de coherencia en el acceso concurrente a los modelos por múltiples aplicaciones independientes, sino en proporcionar acceso simultáneo y eficiente a los datos de muchos modelos desde el código de la aplicación.

6 Anexos

6.1 Siglas y acrónimos

A	
ADM	Architecture Driven Modernization
AFDX	Avionics Full DupleX Switched Ethernet
Alf	Action Language for UML
AM3	AtlanMod MegaModel Management
AMMA	ATLAS Model Management Architecture
AMW	ATLAS Model Weaver
API	Application Programming Interface
ARINC 664	
AS	Abstract Syntax
ASM	Abstract Syntax Model
ASTM	Abstract Syntax Tree Metamodel
ATL	ATLAS Transformation Language
ATP	ATLAS Technical Projector

B	
BBDD	Bases de datos
BNF	Backhus-Naur Format
BPM	Bussiness Process Management
BPMN	Bussiness Process Management Notation

C	
CASE	Computer Aided Software Engineering
CBSE	Component Based Software Engineering
CIM	Computing Independent Model
CMOF	Complete MOF
CS	Concrete Syntax
CSM	Concrete Syntax Model
CWM	Common Warehouse Metamodel

D	
DBMS	Data Base Management System
DDD	Domain Driven Design
DRES	Distributed Real-Time Embedded System

DSD	Domain Specific Development
DS(M)L	Domain Specific (Modeling) Language
DSM	Domain Specific Modeling

E	
EBNF	Extended BNF
EDF	Earliest Deadline First
EMF	Eclipse Modeling Framework
EMP	Eclipse Modeling Project
EMOF	Essential MOF
ETL	Epsilon Transformation Language

F

G	
GCS	Graphical Concrete Syntax
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GP(M)L	General Purpose (Modeling) Language
GUI	Graphical User Interface

H	
HOT	Higher-Order Transformation
HUTN	Human Usable Textual Notation

I	
IDE	Integrated Development Environment
IT	Information Technology

J	
JET	Java Emitter Templates
JMI	Java Metadata Interface
JTL	Janus Transformation Language
JVM	Java Virtual Machine

K	
KDM	Knowledge Discovery Meta-model

L	

M	
M2M	Model to Model
M2T	Model to Text
MARTE	Modeling and Analysis of Real Time and Embedded systems
MAST	Modeling and Analysis Suite for Real-Time Systems
MBE	Model Based Engineering
MDA	Model Driven Architecture
MD(S)D	Model Driven (Software) Development
MD(S)E	Model Driven (Software) Engineering
MDI	Model-Driven Interoperability
MDM	Model Driven Modernization
MDPE	Model Driven Product Engineering
MDRE	Model Driven Reverse Engineering
ML	Modeling Language
MML	Meta-Modeling Language
MOF	Meta-Object Facility
MOFM2T	MOF Model to Text
MTL	Model Transformation Language

N	

O	
OCL	Object Constraint Language
OMG	Object Management Group
OO	Object Oriented

P	
PDM	Platform Description Model
PIM	Platform Independent Model
PSM	Platform Specific Model

Q	
QoS	Quality of Service

QVT	Query / View / Transformation
QVT OML	QVT Operational Mapping Language

R	
RMA	Rate Monotonic Analysis
RTA	Response Time Analysis
RTSJ	Real-Time Specification for Java
RTEP	Real-Time Ethernet Protocol
RubyTL	Ruby Transformation Language

S	
SAM	Schedulabilty Analysis Model
SE	Software Engineering
SLE	Software Language Engineering
SRP	Stack Resource Protocol
SPT	Schedulability, Performance and Time
SDTRE	Sistema Distribuido de Tiempo Real Embebido
SysML	Systems Modeling Language

T	
T2M	Text to Model
TCS	Textual Concrete Syntax
TDD	Test Driven Development
TGG	Triple Graph Grammars
TMF	Textual Modeling Framework
TS	Technical/Technological Space

U	
UML	Unified Modeling Language
URI	Uniform Resource Identifier
UTP	UML Testing Profile

V	
VHDL	VHSIC Hardware Description Language

W	
W3C	World Wide Web Consortium

X	
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSLT	eXtensible Stylesheet Language Transformation
xUML	eXecutable UML

7 Referencias bibliográficas

[1] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering,* 2007, pp. 37-54.

[2] "formal/2011-08-05: Unified Modeling Language (UML) - Infrastructure, v2.4.1," 2011.

[3] "formal/2011-08-06: Unified Modeling Language (UML) - Superstructure, v2.4.1," 2011.

[4] "formal/2012-06-01: Systems Modeling Language (SysML), v1.3," 2012.

[5] "formal/2011-06-02: UML Profile for MARTE: Modeling and Analysis of Real-time Embedded Systems, v1.1," 2011.

[6] J. Bézivin, "On the unification power of models," *Software and Systems Modeling,* vol. 4, pp. 171-188, 2005.

[7] K. Balasubramanian, "Model-Driven Engineering of Component-Based Distributed, Real-Time and Embedded Systems," 2007.

[8] "formal/2005-01-02: UML Profile for Schedulability, Performance, and Time, v1.1," 2005.

[9] J. Bézivin, "In Search of a Basic Principle for Model-Driven Engineering," *Novatica Journal, Special Issue,* pp. 21-24, 2004.

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

[11] E. Seidewitz, "What models mean," *Software, IEEE,* vol. 20, pp. 26-32, 2003.

[12] S. Stavru, I. Krasteva and S. Ilieva, "Challenges of model-driven modernization. an agile perspective." in *MODELSWARD,* 2013, pp. 219-230.

[13] W. Ulrich, "A status on OMG architecture-driven modernization task force," in *Proceedings EDOC Workshop on Model-Driven Evolution of Legacy Systems (MELS). IEEE Computer Society Digital Library,* 2004, .

[14] W. M. Ulrich and P. Newcomb, *Information Systems Transformation: Architecture-Driven Modernization Case Studies.* Morgan Kaufmann, 2010.

[15] G. Blair, N. Bencomo and R. B. France, "Models@run.time," *Computer,* vol. 42, pp. 22-27, 2009.

[16] J. D. Poole, "Model-driven architecture: Vision, standards and emerging technologies," in *Workshop on Metamodeling and Adaptive Object Models, ECOOP,* 2001, .

[17] A. Kleppe, J. Warmer, W. Bast and M. Explained, *The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2003.

[18] J. Bézivin, S. Gérard, P. A. Muller and L. Rioux, "MDA components: Challenges and opportunities," in Workshop on Metamodelling for MDA, York, England, 2003, .

[19] J. Greenfield and K. Short, "Software factories: Assembling applications with patterns, models, frameworks and tools," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications,* 2003, pp. 16-27.

[20] A. Van Deursen, P. Klint and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM Sigplan Notices,* vol. 35, pp. 26-36, 2000.

[21] B. Langlois, C. Jitia and E. Jouenne, "DSL classification," in *OOPSLA 7th Workshop on Domain Specific Modeling,* 2007, .

[22] T. Kühne, "Matters of (meta-) modeling," *Software and Systems Modeling,* vol. 5, pp. 369-385, 2006.

[23] J. Sprinkle, B. Rumpe, H. Vangheluwe and G. Karsai, "Metamodelling. state of the art and research challenges," in *Model-Based Engineering of Embedded Real-Time Systems*Anonymous Springer, 2011, pp. 57-76.

[24] "formal/2014-04-03: Meta Object Facility (MOF) Core, v2.4.2," 2014.

[25] S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development," *Software, IEEE,* vol. 20, pp. 42-45, 2003.

[26] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture,* 2003, pp. 1-17.

[27] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Syst J,* vol. 45, pp. 621-645, 2006.

[28] T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science,* vol. 152, pp. 125-142, 2006.

[29] "formal/2015-02-01 (Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.2)," 2015.

[30] I. Kurtev, "State of the art of QVT: A model transformation language standard," *Applications of Graph Transformations with Industrial Relevance,* pp. 377-393, 2008.

[31] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev and P. Valduriez, "ATL: A QVT-like transformation language," in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications,* 2006, pp. 719-720.

[32] F. Allilaire, J. Bézivin, F. Jouault and I. Kurtev, "ATL: Eclipse support for model transformation," in *Proceedings of the Eclipse Technology eXchange Workshop (eTX) at the ECOOP 2006 Conference, Nantes, France,* 2006, .

[33] F. Jouault, F. Allilaire, J. Bézivin and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming,* vol. 72, pp. 31-39, 2008.

[34] D. Kolovos, R. Paige and F. Polack, "The epsilon transformation language," *Theory and Practice of Model Transformations,* pp. 46-60, 2008.

[35] S. Burmester, H. Giese and W. Schäfer, "Model-driven architecture for hard real-time systems: From platform independent models to code," in *Model Driven Architecture–Foundations and Applications,* 2005, pp. 25-40.

[36] F. Terrier and S. Gérard, "MDE benefits for distributed, real time and embedded systems," in *From Model-Driven Design to Resource Management for Distributed Embedded Systems*Anonymous Springer, 2006, pp. 15-24.

[37] M. U. Khan, K. Geihs, F. Gutbrodt, P. Gohner and R. Trauter, "Model-driven development of real-time systems with UML 2.0 and C," in *Model-Based Development of Computer-Based Systems and Model-Based Methodologies for Pervasive and Embedded Software, 2006. MBD/MOMPES 2006. Fourth and Third International Workshop On,* 2006, pp. 10 pp.-42.

[38] Y. Kacem, A. Mahfoudhi, W. Karamti and M. Abid, "A model driven engineering based method for scheduling analysis," in *Design and Test Workshop, 2008. IDT 2008. 3rd International,* 2008, pp. 326-330.

[39] M. Bordin, M. Panunzio and T. Vardanega, "Fitting schedulability analysis theory into model-driven engineering," in *Real-Time Systems, 2008. ECRTS'08. Euromicro Conference On,* 2008, pp. 135-144.

[40] M. A. Wehrmeister, "An aspect-oriented model-driven engineering approach for distributed embedded real-time systems," in Anonymous University of Paderborn, 2009, .

[41] H. Perez, J. J. Gutierrez, E. Asensio, J. Zamorano and de la Puente, Juan Antonio, "Model-driven development of high-integrity distributed real-time systems using the end-to-end flow model," in *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference On,* 2011, pp. 209-216.

[42] W. E. H. Chehade, A. Radermacher, F. Terrier, B. Selic and S. Gérard, "A model-driven framework for the development of portable real-time embedded systems," in *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference On,* 2011, pp. 45-54.

[43] J. Krueger, E. Engstrom and J. Ward, "MetaDoME: A Rapid Prototyping Tool Supporting Graphical Modeling Tool Development," 1995.

[44] E. Engstrom and J. Krueger, "Building and rapidly evolving domain-specific tools with DOME," in *Computer-Aided Control System Design, 2000. CACSD 2000. IEEE International Symposium On,* 2000, pp. 83-88.

[45] A. Alderson, "Meta-CASE technology," in *Software Development Environments and CASE Technology*Anonymous Springer, 1991, pp. 81-91.

[46] A. Alderson and A. Elliott, "The eclipse tool-builder's kit and the HOOD toolset," in *Proceedings of the Fourth Conference on Software Engineering Environments: Research and Practice,* 1989, pp. 87-109.

[47] K. Smolander, K. Lyytinen, V. Tahvanainen and P. Marttiin, "MetaEdit: A flexible graphical environment for methodology modelling," in *Proceedings of the Third International Conference on Advanced Information Systems Engineering,* 1991, pp. 168-193.

[48] K. Smolander, "OPRR: a model for modelling systems development methods," *Next Generation CASE Tools,* 1991.

[49] T. Klein, U. Nickel, J. Niere and A. Zündorf, "From UML to Java and back again," *University of Paderborn, Tech.Rep,* 1999.

[50] U. Nickel, J. Niere and A. Zündorf, "The FUJABA environment," in *Proceedings of the 22nd International Conference on Software Engineering,* 2000, pp. 742-745.

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

[52] D. Steinberg, F. Budinsky, M. Paternostro and E. Merks, *EMF: Eclipse Modeling Framework.* Addison-Wesley Longman, Amsterdam, 2nd revised edition (rev). edition, 2009.

[53] J. Tolvanen, "MetaEdit+ : Integrated modeling and metamodeling environment for domain-specific languages," in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications,* 2006, pp. 690-691.

[54] J. Tolvanen, R. Pohjonen and S. Kelly, "Advanced tooling for domain-specific modeling: MetaEdit+," in *Sprinkle, J., Gray, J., Rossi, M., Tolvanen, JP (Eds.) the 7th OOPSLA Workshop on Domain-Specific Modeling, Finland,* 2007, .

[55] J. Tolvanen and S. Kelly, "MetaEdit+ : Defining and using integrated domain-specific modeling languages," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications,* 2009, pp. 819-820.

[56] J. Sztipanovits, G. Karsai, C. Biegl, T. Bapty, Á Lédeczi and A. Misra, "MULTIGRAPH: An architecture for model-integrated computing," in *Engineering of Complex Computer Systems, 1995. Held Jointly with 5th CSESAW, 3rd IEEE RTAW and 20th IFAC/IFIP WRTP, Proceedings., First IEEE International Conference On,* 1995, pp. 361-368.

[57] G. Nordstrom, J. Sztipanovits, G. Karsai and A. Ledeczi, "Metamodeling-rapid design and evolution of domain-specific modeling environments," in *Engineering of Computer-Based Systems, 1999. Proceedings. ECBS'99. IEEE Conference and Workshop On,* 1999, pp. 68-74.

[58] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle and P. Volgyesi, "The generic modeling environment," in *Workshop on Intelligent Signal Processing, Budapest, Hungary,* 2001, .

[59] S. Burmester, H. Giese, J. Niere, M. Tichy, J. P. Wadsack, R. Wagner, L. Wendehals and A. Zündorf, "Tool integration at the meta-model level: the Fujaba approach," *International Journal on Software Tools for Technology Transfer,* vol. 6, pp. 203-218, 2004.

[60] C. Amelunxen, A. Königs, T. Rötschke and A. Schürr, "MOFLON: A standard-compliant metamodeling framework with graph transformations," in *Model Driven Architecture–Foundations and Applications,* 2006, pp. 361-375.

[61] E. Kindler and R. Wagner, "Triple graph grammars: Concepts, extensions, implementations, and application scenarios," *University of Paderborn,* 2007.

[62] B. Demuth, "The dresden OCL toolkit and its role in information systems development," in *Proc. of the 13th International Conference on Information Systems Development (ISD'2004),* 2004, .

[63] A. Anjorin, M. Lauder, S. Patzina and A. Schürr, "eMoflon: Leveraging EMF and Professional CASE Tools," *Informatik,* pp. 281, 2011.

[64] P. Swithinbank, M. Chessell, T. Gardner, C. Griffin, J. Man, H. Wylie and L. Yusuf, *Patterns: Model-Driven Development using IBM Rational Software Architect.* IBM, International Technical Support Organization, 2005.

[65] D. Leroux, M. Nally and K. Hussey, "Rational Software Architect: A tool for domain-specific modeling," *IBM Syst J,* vol. 45, pp. 555-568, 2006.

[66] Y. Guo, K. Sierszecki and C. Angelov, "Model-driven development of domain-specific applications: Tool support," in *Proc. of the 7th Nordic Workshop on Model Driven Software Engineering NW-MODE,* 2009, pp. 225-239.

[67] Y. Guo, K. Sierszecki and C. K. Angelov, "COMDES development toolset," in *The 5th International Workshop on Formal Aspects of Component Software FACS 2008,* 2008, pp. 233-237.

[68] A. El Kouhen, C. Dumoulin, S. Gérard and P. Boulet, "Evaluation of Modeling Tools Adaptation," 2012.

[69] H. Kern, A. Hummel and S. Kühne, "Towards a comparative analysis of meta-metamodels," in *Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11,* 2011, pp. 7-12.

[70] V. Dimitrieski, M. Celikovic, V. Ivancevic and I. Lukovic, "A comparison of ecore and GOPPRR through an information system meta modeling approach," in *8th European Conference on Modelling Foundations and Applications (ECMFA), Technical University of Denmark, Joint Proceedings, ISBN,* 2012, pp. 978-987.

[71] J. Bézivin, C. Brunette, R. Chevrel, F. Jouault and I. Kurtev, "Bridging the generic modeling environment (GME) and the eclipse modeling framework (EMF)," in *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA,* 2005, .

[72] J. Bézivin, G. Hillairet, F. Jouault, I. Kurtev and W. Piers, "Bridging the MS/DSL tools and the eclipse modeling framework," in *Proceedings of the International Workshop on Software Factories at OOPSLA,* 2005, .

[73] J. Bézivin, F. Jouault and D. Touzet, "An introduction to the ATLAS Model Management Architecture," *Research Report LINA,(05-01),* 2005.

[74] H. Kern, "The interchange of (meta) models between metaedit and eclipse emf using m3-level-based bridges," in *8th OOPSLA Workshop on Domain-Specific Modeling at OOPSLA,* 2008, .

[75] D. C. Schmidt, A. Gokhale, B. Natarajan, S. Neema, T. Bapty, J. Parsons, J. Gray, A. Nechypurenko and N. Wang, "CoSMIC: An MDA generative tool for distributed real-time and embedded component middleware and applications," in *OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture, Seattle, WA,* 2002, .

[76] A. Childs, J. Greenwald, G. Jung, M. Hoosier and J. Hatcliff, "Calm and cadena: Metamodeling for component-based product-line development," *Computer,* vol. 39, pp. 42-50, 2006.

[77] P. H. Feiler, D. P. Gluch and J. J. Hudak, "The architecture analysis & design language (AADL): An introduction," *The Architecture Analysis & Design Language (AADL): An Introduction,* 2006.

[78] P. Farail, P. GOUTILLET, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut and M. Pantel, "The TOPCASED project: a toolkit in open source for critical aeronautic systems design," *Ingenieurs De L'Automobile,* pp. 54-59, 2006.

[79] É. Conquet, M. Perrotin, P. Dissaux, T. Tsiodras and J. Hugues, "The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software." 2010.

[80] M. Perrotin, E. Conquet, J. Delange, A. Schiele and T. Tsiodras, "TASTE: A real-time software engineering tool-chain overview, status, and future," in *SDL 2011: Integrating System and Software Modeling*Anonymous Springer, 2012, pp. 26-37.

[81] M. Perrotin, E. Conquet, J. Delange and T. Tsiodras, "TASTE: An open-source tool-chain for embedded system and software development," in *Proceedings of the Embedded Real Time Software and Systems Conference (ERTS), Toulouse, France,* 2012, .

[82] S. Burmester, H. Giese, M. Hirsch, D. Schilling and M. Tichy, "The fujaba real-time tool suite: Model-driven development of safety-critical, real-time systems," in *Proceedings of the 27th International Conference on Software Engineering,* 2005, pp. 670-671.

[83] S. Becker, S. Dziwok, T. Gewering, C. Heinzemann, U. Pohlmann, C. Priesterjahn, W. Schäfer, O. Sudmann and M. Tichy, "MechatronicUML - Syntax and Semantics," *Software Engineering Group, Heinz Nixdorf Institute, Number Tr-Ri-11-325,* 2011.

[84] S. E. Wedin, "Model-driven architecture and xtuml in practice," in *ESF Conference,* 2009, .

[85] "Description of the MAST Model," .

[86] "Analysis Techniques used in MAST," .

[87] P. Herzum and O. Sims, *Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise.* John Wiley & Sons, Inc., 2000.

[88] P. López Martínez, C. Cuevas and J. M. Drake, "Model-driven design of real-time component-based applications," in *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference On,* 2010, pp. 1-8.

[89] L. Abeni and G. Buttazzo, "Resource Reservation in Dynamic Real-Time Systems," *Real-Time Syst.,* vol. 27, pp. 123-167, 2004.

[90] L. Barros, C. Cuevas, P. López Martínez, J. M. Drake and M. González Harbour, "Modelling real-time applications based on resource reservations," *Journal of Systems Architecture, Elsevier,* 2013.

[91] P. Lopez Martinez, J. M. M. Lanza, J. M. Drake and M. Gonzalez Harbour, "Framework for the design of java firm real-time systems oriented to the generation of timing behaviour models," in *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference On,* 2013, pp. 195-202.

[92] 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.

[93] I. Kurtev, J. Bézivin and M. Akşit, "Technological Spaces: An initial appraisal," 2002.

[94] J. Bézivin, H. Bruneliere, F. Jouault and I. Kurtev, "Model engineering support for tool interoperability," in *Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005), Montego Bay, Jamaica,* 2005, .

[95] J. Bézivin and I. Kurtev, "Model-based technology integration with the technical space concept," in *Metainformatics Symposium,* 2005, .

[96] J. Bézivin, "Model Driven Engineering: An Emerging Technical Space," *Generative and Transformational Techniques in Software Engineering,* pp. 36-64, 2006.

[97] F. S. Parreiras, S. Staab and A. Winter, "On marrying ontological and metamodeling technical spaces," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering,* 2007, pp. 439-448.

[98] J. Bézivin, V. Devedzic, D. Djuric, J. M. Favreau, D. Gasevic and F. Jouault, "An M3-Neutral infrastructure for bridging model engineering and ontology engineering," *Interoperability of Enterprise Software and Applications,* pp. 159-171, 2006.

[99] J. Bézivin, R. M. Soley and A. Vallecillo, "Model-driven interoperability: MDI 2010," in *Models in Software Engineering*Anonymous Springer, 2011, pp. 145-149.

[100] "formal/2014-02-03: Object Constraint Language (OCL), v2.4," 2014.

[101] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA.* Addison-Wesley Longman Publishing Co., Inc., 2003.

[102] E. Miliauskaite and L. Nemuraite, "Taxonomy of integrity constraints in conceptual models," in *IADIS Virtual Multi Conference on Computer Science and Information Systems,* 2005.

[103] E. Miliauskaite and L. Nemuraite, "Representation of integrity constraints in conceptual models," *Information Technology and Control,* vol. 34, pp. 355-365, 2005.

[104] R. S. Scowen, "Extended BNF-a generic base standard," 1998.

[105] H. Grönninger, H. Krahn, B. Rumpe, M. Schindler and S. Völkel, "Text-based Modeling," *arXiv Preprint arXiv:1409.6623,* 2014.

[106] M. Fowler, "Language Workbenches: The Killer-App For Domain Specific Languages?" 2005.

[107] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly and A. Loh, "The state of the art in language workbenches," in *Software Language Engineering*Anonymous Springer, 2013, pp. 197-217.

[108] S. Kelly, "Empirical comparison of language workbenches," in *Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling,* 2013, pp. 33-38.

[109] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion,* 2010, pp. 307-309.

[110] F. Jouault, J. Bézivin and I. Kurtev, "TCS:: A DSL for the specification of textual concrete syntaxes in model engineering," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering,* 2006, pp. 249-254.

[111] P. Charles, R. M. Fuhrer and S. M. Sutton Jr, "IMP: A meta-tooling platform for creating language-specific IDEs in eclipse," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering,* 2007, pp. 485-488.

[112] M. Scheidgen, "Integrating content assist into textual modelling editors." in *Modellierung,* 2008, pp. 14.

[113] F. Heidenreich, J. Johannes, S. Karol, M. Seifert and C. Wende, "Model-based language engineering with EMFText," in *Generative and Transformational Techniques in Software Engineering IV*Anonymous Springer, 2013, pp. 322-345.

[114] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler and S. Völkel, "Monticore: A framework for the development of textual domain specific languages," in *Companion of the 30th International Conference on Software Engineering,* 2008, pp. 925-926.

[115] M. Voelter and K. Solomatov, "Language modularization and composition with projectional language workbenches illustrated with MPS," *Software Language Engineering, SLE,* vol. 16, 2010.

[116] L. Iovino, A. Pierantonio and I. Malavolta, "On the Impact Significance of Metamodel Evolution in MDE." *Journal of Object Technology,* vol. 11, pp. 3:1-33, 2012.

[117] H. Bruneliere, J. Garcia, P. Desfray, D. E. Khelladi, R. Hebig, R. Bendraou and J. Cabot, "On lightweight metamodel extension to support modeling tools agility," in *11th European Conference on Modelling Foundations and Applications (ECMFA 2015)(a STAF 2015 Conference),* 2015, .

[118] D. Di Ruscio, L. Iovino and A. Pierantonio, "Managing the coupled evolution of metamodels and textual concrete syntax specifications," in *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference On,* 2013, pp. 114-121.

[119] A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Retschitzegger, J. Schoenboeck, W. Schwinger and M. Wimmer, "A systematic taxonomy of metamodel evolution impacts on OCL expressions," in 2014, pp. 2.

[120] A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Retschitzegger, J. Schoenboeck, W. Schwinger and M. Wimmer, "Systematic Co-Evolution of OCL Expressions," *11th APCCM,* vol. 27, pp. 30, 2015.

[121] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *ECOOP 2007–Object-Oriented Programming*Anonymous Springer, 2007, pp. 600-624.

[122] B. Gruschko, D. Kolovos and R. Paige, "Towards synchronizing models with evolving metamodels," in *Proceedings of the International Workshop on Model-Driven Software Evolution,* 2007, .

[123] A. Cicchetti, D. D. Ruscio, R. Eramo and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE,* 2008, pp. 222-231.

[124] L. M. Rose, R. F. Paige, D. S. Kolovos and F. A. Polack, "An analysis of approaches to model migration," in *Proc. Joint MoDSE-MCCM Workshop,* 2009, pp. 6-15.

[125] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev and A. Lindow, "Model transformations? Transformation models!" *Model Driven Engineering Languages and Systems,* pp. 440-453, 2006.

[126] M. Tisi, F. Jouault, P. Fraternali, S. Ceri and J. Bézivin, "On the use of higher-order model transformations," in *Model Driven Architecture-Foundations and Applications,* 2009, pp. 18-33.

[127] M. Tisi, J. Cabot and F. Jouault, "Improving Higher-Order Transformations support in ATL," *Theory and Practice of Model Transformations,* pp. 215-229, 2010.

[128] J. Bézivin and F. Jouault, "Using ATL for checking models," *Electronic Notes in Theoretical Computer Science,* vol. 152, pp. 69-81, 2006.

[129] J. L. Diguet, "Checking syntactic constraints on models using ATL model transformations," *Model Transformation with ATL,* pp. 140, 2009.

[130] M. Elaasar, L. Briand and Y. Labiche, "Domain-specific model verification with QVT," in *Modelling Foundations and Applications*Anonymous Springer, 2011, pp. 282-298.

[131] X. Oriol and E. Teniente, "Incremental checking of OCL constraints through SQL queries," in *CEUR Workshop Proceedings,* 2014, pp. 23-32.

[132] R. Delmas, A. F. Pires and T. Polacsek, "A verification and validation process for model driven engineering," in *Progress in Flight Dynamics, Guidance, Navigation, Control, Fault Detection, and Avionics,* 2013, pp. 455-468.

[133] K. Anastasakis, B. Bordbar, G. Georg and I. Ray, "UML2Alloy: A challenging model transformation," in *Model Driven Engineering Languages and Systems*Anonymous Springer, 2007, pp. 436-450.

[134] J. Cabot, R. Clarisó and D. Riera, "UMLtoCSP: A tool for the formal verification of UML/OCL models using constraint programming," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering,* 2007, pp. 547-548.

[135] C. A. G. Pérez, F. Buettner, R. Clarisó and J. Cabot, "EMFtoCSP: A tool for the lightweight verification of EMF models," in *Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA),* 2012, .

[136] J. Lehoczky, L. Sha and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Real Time Systems Symposium, 1989., Proceedings.* 1989, pp. 166-171.

[137] G. Wiederhold, *Views, Objects, and Databases.* Springer, 1991.

[138] H. Bruneliere, J. G. Perez, M. Wimmer and J. Cabot, "EMF views: A view mechanism for integrating heterogeneous models," in *34th International Conference on Conceptual Modeling (ER 2015),* 2015, .

[139] E. Burger, "Flexible views for rapid model-driven development," in *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling,* 2013, pp. 1.

[140] A. Cicchetti, F. Ciccozzi and T. Leveque, "A hybrid approach for multi-view modeling," *Electronic Communications of the EASST,* vol. 50, 2012.

[141] D. Bork, D. I. Karagiannis and H. I. Fill, "Model-Driven Development of Multi-View Modelling Tools: The MuVieMoT Approach," 2014.

[142] SAE, “Requirements Definition and Analysis Language” (Draft), Available at https://wiki.sei.cmu.edu/aadl/images/3/37/RDAL_annex_draft_v16.pdf