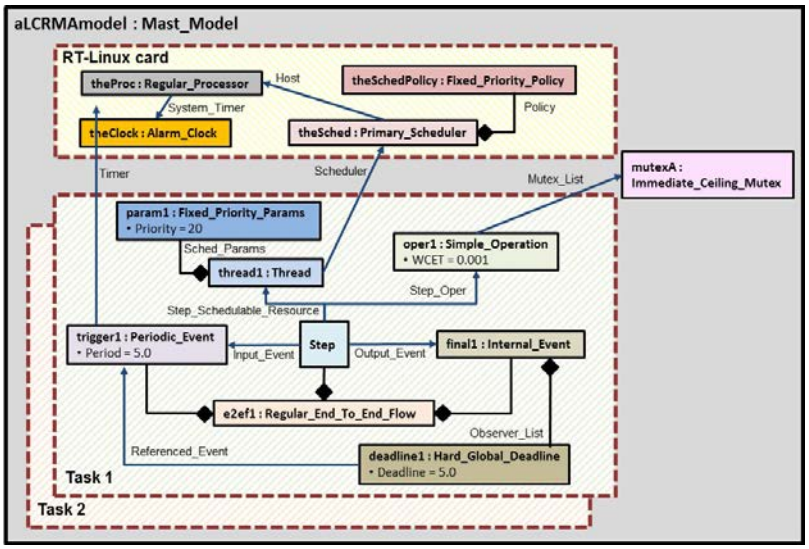

Fig. 5. Ejemplo de modelo LinuxClassicRMA

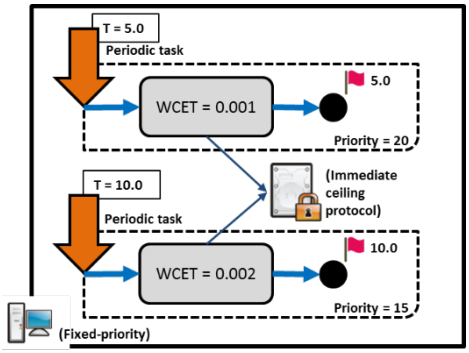

Fig. 6. Visión reactiva del ejemplo de modelo LinuxClassicRMA

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 125 clases de MAST 2.0, sino incluso de tener que conocer las 17 clases que se necesitan para modelar en el entorno MAST 2.0 el *software* con las restricciones de la empresa.

Todo el material relativo al ejemplo LinuxClassicRMA, incluyendo su especificación exhaustiva, puede encontrarse en [10]. Como para cualquier vista, toma como base el meta-modelo cuya estructura se desea restringir (MAST 2.0) 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, establecimiento de referencias a *null*, etc.) y estipula los elementos que obligatoriamente han de existir en todo modelo acorde a la vista.

Meta-modelo LinuxClassicRMA_VRD. El diagrama de clases mostrado en la Fig. 7 representa el meta-modelo VRD correspondiente a la vista LCRMA. El diseñador que afronta la tarea de crear modelos LCRMA únicamente debe construir modelos conformes a este meta-modelo reducido, los cuales posteriormente serán transformados a modelos MAST 2.0 (acordes a LCRMA) mediante la transformación LinuxClassicRMA_VRD_to_MAST2 (particularización de VRD_To_Domain de la Fig. 4).

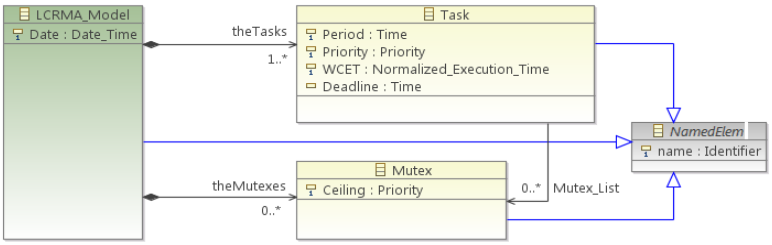

Fig. 7. Meta-modelo LinuxClassicRMA_VRD

La Fig. 8 muestra el modelo que ha de construir el diseñador para obtener el modelo de la Fig. 5. La diferencia de complejidad conceptual y tamaño es sustancial.

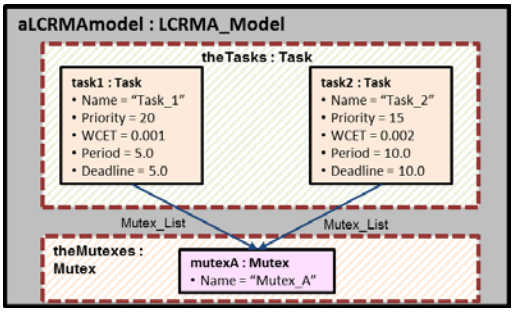

Fig. 8**.** Modelo VRD construido por el diseñador

3.4 Meta-modelo CVS para especificación de vistas restrictivas

Se presenta una visión general de meta-modelo para Especificación de Vistas Restrictivas (*Constraining View Specification*, CVS), empleando Ecore como lenguaje de meta-modelado. El diagrama de clases de la Fig. 9 muestra dicho meta-modelo, 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 meta-modelo 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 meta-modelo 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.

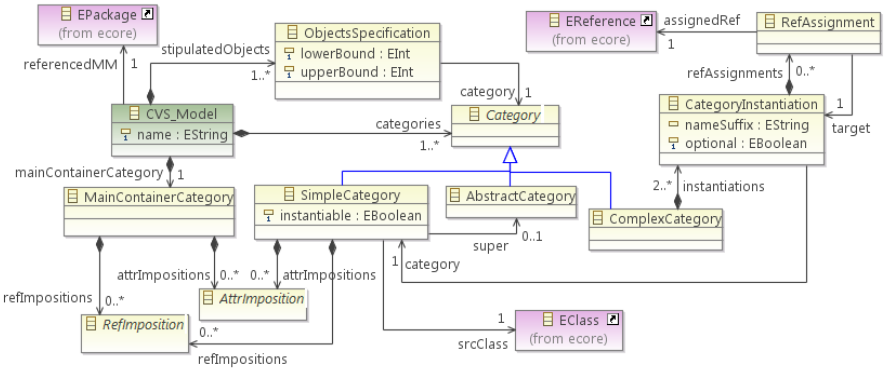

Fig. 9. Meta-modelo CVS

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 meta-modelo CVS define tres subclases de *Category*.

- ***SimpleCategory***. Representa el concepto de *categoría básica definida por una vista sobre una clase (permitida) del correspondiente meta-modelo*. 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 meta-modelo de dominio. En su lugar, el meta-modelo 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 meta-herramienta expuesta en la Sección 4.

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 Fig. 10 muestra las clases mencionadas

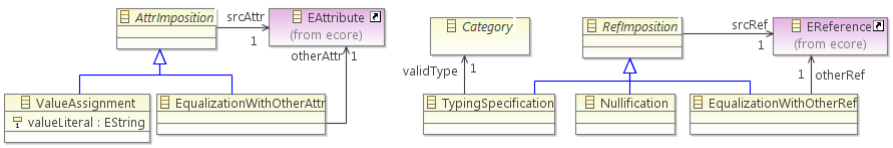

Fig. 10. Clases relativas a imposiciones sobre atributos y sobre referencias

La vista LinuxClassicRMA formulada como modelo CVS. La Fig. 11 muestra una parte del modelo CVS representativo de la vista LinuxClassicRMA. Por razones de espacio no se expone aquí en su totalidad la visualización gráfica del modelo completo. En su lugar, la figura 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 de prioridades fijas. El ensamblado *RT_Lin_Card* se define mediante 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 meta-modelo MAST 2.0. 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.

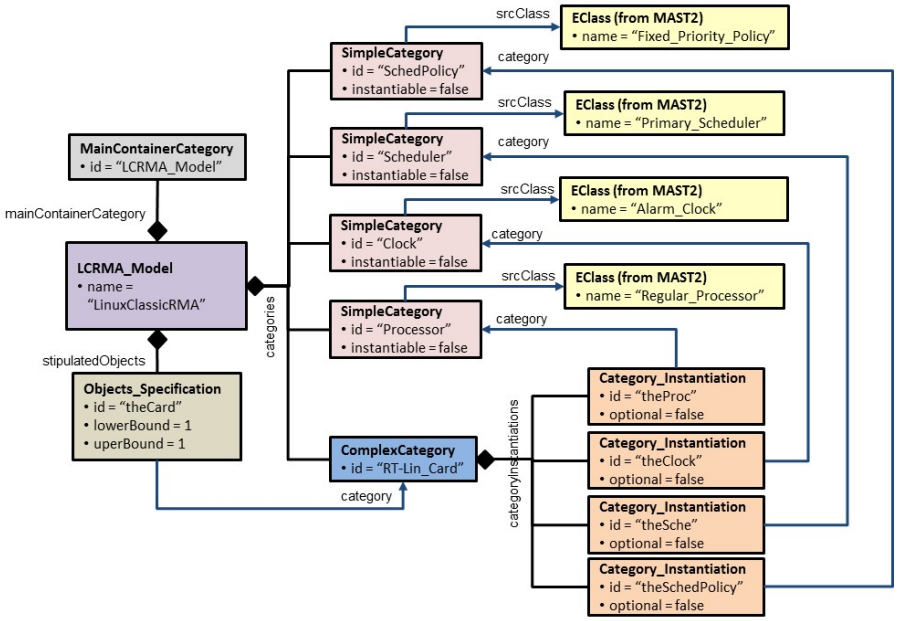

Fig. 11. Parte del modelo CVS representativo de la vista LinuxClassicRMA

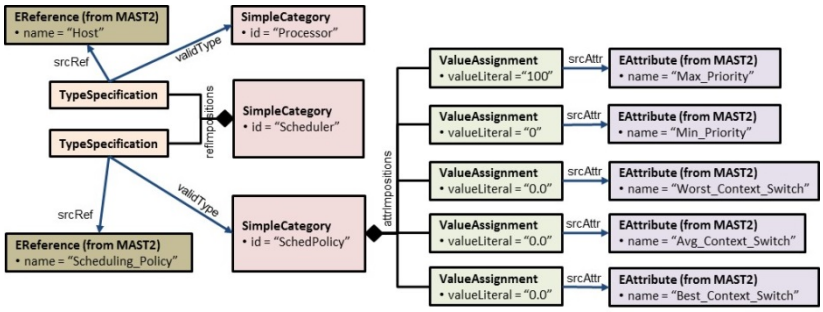

Fig. 12. Detalle de las categorías simples *Scheduler* y *SchedPolicy*

La Fig. 12 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.

4 Herramienta genérica para construcción de modelos

En la sección anterior se ha expuesto el diseño de una herramienta para facilitar la construcción de modelos acordes a una vista, basada en una estrategia que requiere desarrollar dos componentes (meta-modelo VRD y transformación VRD_to_Domain) propios de la vista en cuestión (y por extensión del meta-modelo de dominio).

En esta sección se expone el diseño de una herramienta genérica, aplicable a cualquier meta-modelo de dominio y a cualquier vista especificada sobre él. Su carácter genérico se consigue en forma de meta-herramienta que genera bajo demanda la herramienta específica correspondiente.

La meta-herramienta opera en dos pasos, generando sucesivamente los dos componentes de cada herramienta específica a partir de la formulación de la vista conforme al meta-modelo CVS expuesto en la subsección 3.4. La Fig. 13 lo esquematiza.

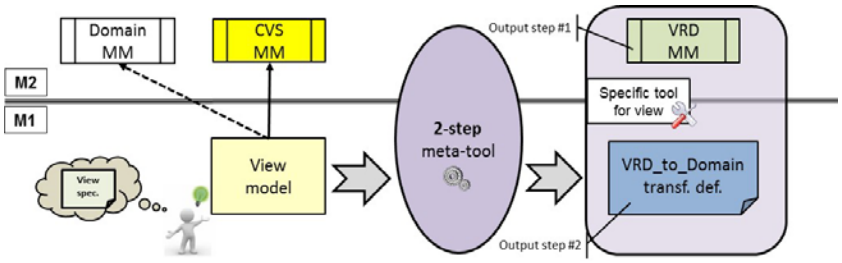

Fig. 13. Meta-herramienta de dos pasos

1. Meta-modelo VRD que dirige la construcción restringida de modelos. Tal y como se muestra en la Fig. 14, este componente se obtiene a partir del modelo CVS a través de una transformación promocionadora (CVS_to_VRD), esto es, una transformación M2M que toma como entrada un artefacto en la capa M1 (modelo) y produce como salida un artefacto en la capa M2 (meta-modelo).

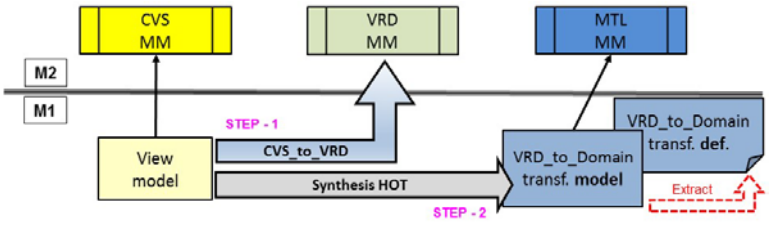

Fig. 14. Generación automática del metamodelo VRD y de la transformación VRD_to_Domain

2. Transformación M2M que convierte los modelos de datos requeridos a modelos conformes al meta-modelo de dominio. Según se muestra en la Fig. 14, a partir del modelo CVS se hace uso de la técnica de Transformaciones de Orden Superior (*Higher Order Transformations*, HOT [11]) para obtener la transformación VRD_to_Domain en forma de modelo conforme al meta-modelo del lenguaje de transformación utilizado. En este trabajo se ha empleado ATL [12, 13], que gracias a tener su sintaxis formalizada como meta-modelo, permite la aplicación de la técnica HOT. 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 meta-modelo VRD generado en el primer paso.

La estrategia desarrollada involucra diversas transformaciones M2M. En primer lugar, dado un caso concreto de vista especificada sobre un meta-modelo de dominio, los modelos reducidos construidos por el diseñador son transformados a los modelos definitivos mediante la correspondiente transformación VRD_to_Domain. En lo relativo a la generación automática de los componentes correspondientes a cada situación (meta-modelo VRD y la propia transformación VRD_to_Domain), se utilizan respectivamente la transformación M2MM (o promocionadora) CVS_To_VRD y la HOT CVS_to_MTL, donde MTL es el lenguaje de transformación empleado (ATL en este caso). La implementación ATL de cada una de ellas puede encontrarse junto al resto del material asociado a este trabajo en [9].

5 Conclusiones y trabajo futuro

Los editores genéricos de que disponen los diseñadores para crear modelos desde un terminal suelen estar asistidos por la información reflexiva que obtienen de sus meta-modelos. En este trabajo se mejoran los casos en que esta estrategia no resulta amigable al diseñador, bien porque el meta-modelo no sigue la lógica que éste espera para introducir los datos, o bien porque el meta-modelo es excesivamente complejo. La solución propuesta es utilizar para la edición un meta-modelo especializado y la conversión posterior del modelo editado al modelo final.

En la versión actual, el método se puede aplicar a cualquier meta-modelo de partida, pero sólo se ha contemplado definir vistas de edición que son útiles para el caso particular de modelos conformes a un meta-modelo de dominio genérico que se restringe reduciendo las clases que se usan, se le modifican las multiplicidades o se introducen nuevas clases que definen determinados patrones de instancias. Un trabajo futuro a realizar es extender la metodología y las herramientas a las necesidades de vistas que se generen en otras situaciones.

Agradecimientos. Este trabajo ha sido financiado parcialmente por el Gobierno de España y fondos FEDER con referencias TIN2011-28567-C03-02 (HI-PARTES) y TIN2014-56158-C4-2-P (M2C2).

Referencias

1 "formal/2012-06-01: Systems Modeling Language (SysML), v1.3," 2012.

2 "formal/2011-06-02: UML Profile for MARTE: Modeling and Analysis of Real-time Embedded Systems, v1.1," 2011.

3 M. González Harbour, J. J. Gutiérrez, J. L. Medina, J. C. Palencia, J. M. Drake, J. M. Rivas, P. López Martínez and C. Cuevas, "MAST: Bringing response-time analysis into real-time systems engineering," in *Proceedings of a Conference Organized in Celebration of Pro-Fessor Alan Burns' Sixtieth Birthday,* pp. 42.

4 G. Wiederhold, *Views, Objects, and Databases.* Springer, 1991.

5 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, .

6 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.

7 A. Cicchetti, F. Ciccozzi and T. Leveque, "A hybrid approach for multi-view modeling," *Electronic Communications of the EASST,* vol. 50, 2012.

8 D. Bork, D. I. Karagiannis and H. I. Fill, "Model-Driven Development of Multi-View Modelling Tools: The MuVieMoT Approach," 2014.

9 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.

10 http://www.istr.unican.es/members/cesarcuevas/phd/constraining_views.html

11 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.

12 F. Jouault, F. Allilaire, J. Bézivin and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming,* vol. 72, pp. 31-39, 2008.

13 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.