PyEmofUC: Un entorno MDE/EMOF minimalista

José M. Drake, César Cuevas, Juan R. Fernández Castañera y Patricia López Martínez

ISTR, Universidad de Cantabria, España

{drakej,cuevasce,lopezpa}@unican.es

Abstract. Se presenta el entorno PyEmofUC para la creación, procesado, transformación y visualización de información en base al paradigma de ingeniería dirigida por modelos (MDE). Los meta-modelos se formulan de acuerdo con la especificación EMOF de la organización OMG y se implementan utilizando el lenguaje de programación Python. El entorno es multiplataforma, abierto y minimalista. Además del espacio tecnológico de modelado nativo, basado en Python y EMOF, el entorno da soporte al espacio tecnológico basado en lenguajes específicos como medio de facilitar la interacción con los expertos de dominio, y al espacio tecnológico de serialización para el almacenamiento persistente de los modelos y para la inter-operación con otros entornos. Por último, PyEmofUC permite formular transformaciones de modelos utilizando estilos imperativo, declarativo e híbrido.

Keywords: MDE, EMOF, meta-modelado, Python.

1 Introducción.

PyEmofUC [1] es un entorno para la creación, procesamiento y visualización de información en base al paradigma de ingeniería dirigida por modelos (MDE), en el que todos los elementos (meta-modelos y modelos) se formulan de acuerdo con la especificación EMOF 2.4 [2] de OMG. El entorno ha sido implementado utilizando el lenguaje Python y con ello se ha conseguido que sea:

- *Multiplataforma*: Opera en cualquier plataforma de ejecución que disponga de un intérprete Python 2.7.
- *Abierto*: Opera sobre cualquier información formalizada de acuerdo a EMOF y que pueda ser accedida desde el entorno.
- *Minimalista*: El entorno es ligero (se distribuye como un paquete Python con sólo siete módulos y un total de 2553 LLoC). Ofrece una interfaz pública sencilla a las aplicaciones que lo usan. Cada modelo puede ser reducido (para su almacenamiento persistente e intercambio con otros entornos) como un texto XMI, y cada transformación M2M se reduce a un *script* de código Python.

El entorno ha sido diseñado con el objetivo de gestionar (leer, crear, procesar, visualizar, intercambiar o almacenar) modelos desde aplicaciones o desde simples

scripts Python que hagan uso del paradigma MDE para representar la información. La aplicación sólo invoca los métodos de la interfaz pública del entorno y no necesita disponer ni conocer ninguna infraestructura de soporte del entorno.

Se ha buscado satisfacer las siguientes características específicas:

- *Espacios tecnológicos*: Tal y como muestra la Fig.1, el entorno PyEmofUC da soporte desde su núcleo a tres espacios tecnológicos:

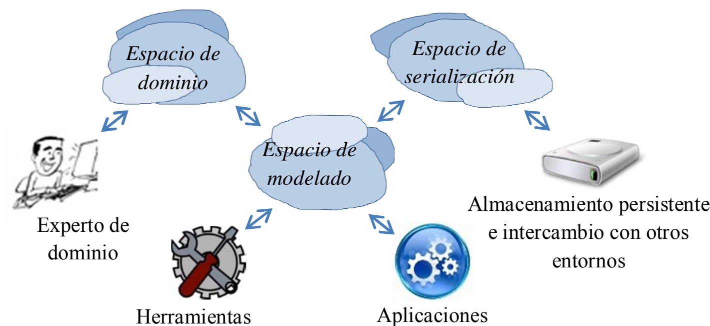

Fig. 1. Espacios tecnológicos que soporta PyEmofUC.

  – *Espacio de modelado*: Es el espacio tecnológico nativo del entorno. La información se representa mediante estructuras de datos Python que implementan fielmente a las establecidas en EMOF. Constituye la formulación básica de la información que es accesible desde los programas.
  – *Espacio de serialización:* La información se formula como texto XMI, y las referencias como URIs textuales estándares [3]. En el entorno se usa para dar soporte al almacenamiento persistente de los modelos en ficheros y servidores *web*, así como para el intercambio de modelos con otros entornos.
  – *Espacio de dominio*: La información se formula mediante un lenguaje específico de dominio que el entorno genera automáticamente a partir del correspondiente meta-modelo. Su finalidad en el entorno es facilitar al experto de dominio la creación y supervisión de los modelos a través de editores asistidos y consolas enriquecidas sin necesidad de tener que ser experto en otras tecnologías MDE.
- *No requiere generación de código complementario*: Una aplicación opera directamente sobre los modelos y para la instanciación de éstos no requiere la generación previa de código fuente de soporte. Las clases que requiere la aplicación para implementar los modelos en memoria de la aplicación se generan dinámicamente durante su ejecución, en base a los datos del modelo y a la interpretación de su meta-modelo. Esto se puede realizar porque en Python las clases y funciones se pueden definir y elaborar en tiempo de ejecución, a través de información textual sencilla.
- Las *transformaciones M2M pueden ser formuladas con estilo imperativo, declarativo o híbrido*: Las transformaciones se formulan como funciones Python, y pueden utilizarse los tres estilos gracias a que Python es a la vez un lenguaje procedural, orientado a objetos y funcional.

- *EmofUC extiende EMOF con nuevas capacidades*: PyEmofUC utiliza el meta-modelo EmofUC que implementa en Python la especificación EMOF 2.4 de OMG. Así mismo, hace uso de los elementos *tag* definidos en EMOF para implementar extensiones relativas a herencia y restricción de rangos de valores en tipos primitivos, definición de las propiedades derivadas y asignación de funcionalidad a las operaciones.
- *Ofrece reflexividad completa:* Todo elemento de un modelo cargado en el entorno tiene siempre una referencia directa a su meta-clase, y a través de ella, a la información que describe propiedades y operaciones. Con ello se facilita el desarrollo de herramientas con capacidad de operar sobre modelos cuyos meta-modelos se definan posteriormente.
- *Todos los elementos contenedores de información son tratados como modelos*: El entorno gestiona de igual forma los meta-modelos (M2) y los modelos (M1). La única excepción es el meta-meta-modelo (M3) que se implementa directamente a partir del código del entorno y es inmutable.

2 Antecedentes y motivación.

Este trabajo tiene como antecedente el proyecto PyEmof realizado en 2007 por Rafael Marvie [4]. El único objetivo que perseguía este antecesor era representar mediante estructuras de datos Python los modelos conformes a meta-modelos EMOF, y con ello, facilitar su transformación utilizando código Python. Su implementación fue incompleta y no ha evolucionado desde 2008.

Actualmente, la tecnología generada por Eclipse Modeling Project (EMP) [5, 6] domina la construcción de entornos MDE. Este proyecto Eclipse cubre toda la metodología MDE, y por ello requiere para el desarrollo de cualquier proyecto la utilización de una cantidad de recursos y conocimientos que no están justificados si el proyecto es pequeño y tiene un alcance limitado. También están disponibles otras plataformas de desarrollo MDE. Algunas de ellas como Microsoft DSL Tools [7, 8] o IBM Rational Software Architect [9] tienen unos objetivos similares a EMF, pero son cerradas. Otras plataformas como Generic Modeling Environment (GME) [10, 11], Fujaba Tool Suite [12] y Obeo Designer [13] aportan nuevas metodologías o enfoques conceptuales, aunque en todas ellas se requiere una infraestructura compleja desde donde se gestionan los modelos.

El planteamiento de este proyecto está en la línea de Marvie y se diferencia de los anteriores en que reduce su objetivo a aportar un núcleo ligero de recursos que puede ser incorporado a las herramientas creadas por los expertos de dominios específicos. A éstos les resulta muy atractivo el uso de la metodología MDE, pero no quieren tener que ser expertos también en las complejas infraestructuras que le dan soporte.

3 Elementos del entorno PyEmofUC

La Fig. 2 muestra como el entorno PyEmofUC estructura la representación de la información según la arquitectura de 3+1 capas de OMG en los tres espacios tecnológicos ya citados.

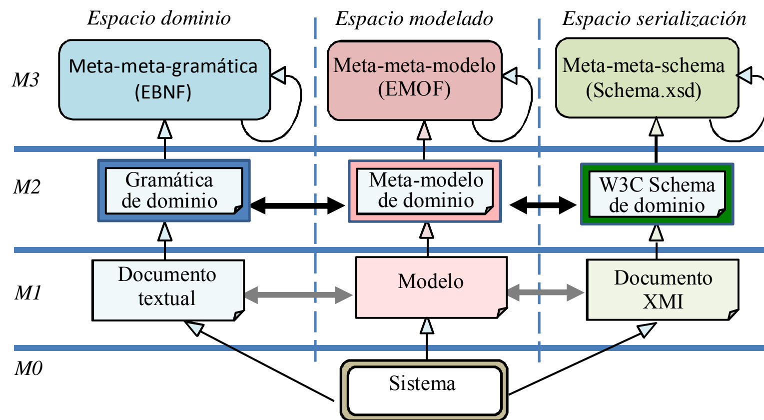

Fig. 2. Capas de meta-modelado y espacios tecnológicos.

PyEmofUC define las estructuras de datos y la interfaz pública para la gestión de los modelos y meta-modelos en los diferentes espacios, así como la interoperatividad entre capas y entre espacios. El entorno no es simétrico respecto de los tres espacios. El espacio de modelado es el nativo del entorno y juega el papel de pivote respecto de los otros espacios, que sólo son utilizados para usos específicos como interacción con los expertos de dominio y cuando el modelo se representa aisladamente y fuera del entorno.

3.1 Recursos del espacio de modelado

En el espacio de modelado la información se representa en la memoria de la aplicación mediante estructuras de datos Python que implementan la especificación EMOF. La información de los meta-modelos pueden interpretarse directamente, ya que son conformes a EMOF, y éste es siempre accesible. La información que contienen los modelos es interpretada en base a sus meta-modelos. Como se muestra en la Fig. 3, el entorno proporciona acceso desde cualquier elemento de modelo al correspondiente elemento del meta-modelo que constituye su meta-clase, y a través de ella, las aplicaciones disponen de una información reflexiva completa sobre ellos. Basándose en esta reflexividad, las aplicaciones que operan sobre los modelos del entorno pueden ser desarrolladas antes de que hayan sido definidos sus meta-modelos.

Aspectos característicos de la implementación del espacio de modelado son:

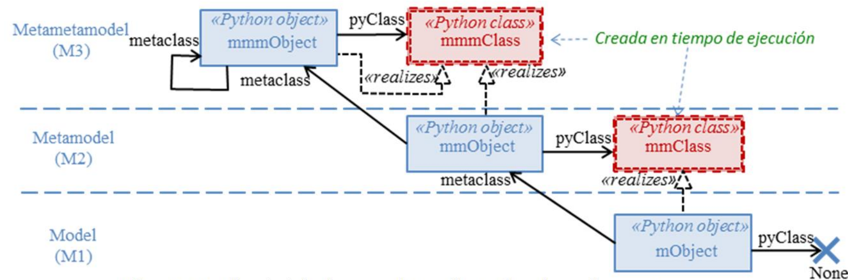

Fig. 3. Reflexividad completa de todos los elementos.

- La información de un modelo (o meta-modelo) se organiza en memoria de la aplicación mediante un conjunto de objetos Python que son instancias de las clases Python que se diseñan con las estructuras de datos y el comportamiento descritos en el objeto que representa su meta-clase en el meta-modelo. Como se muestra en la Fig. 3, cada elemento de un nivel de modelado está compuesto por dos estructuras:
  - Una estructura de datos (*«python object»*) con capacidad de contener la información del elemento, y que es una instancia de la clase asociada a la meta-clase en su meta-modelo.
  - Una clase Python (*«python class»*) que es elaborada en fase de ejecución a partir de la información contenida en la estructura previa, y cuyas instancias son las estructuras de datos de los objetos del nivel inferior.

  Lo característico en PyEmofUC es que, gracias a las facilidades que proporciona Python, las clases *«python class»* son también estructuras de datos que se crean dinámicamente en tiempo de ejecución a partir del objeto Python del mismo nivel.
- En el espacio de modelado todas las referencias entre objetos (entre objetos de un mismo modelo, de un objeto de un modelo a un objeto de otro modelo), o de un modelo con su metamodelo se implementan mediante referencias Python estándares. Esto requiere que todos los modelos referenciados desde los objetos de un modelo deben estar también instanciados en la memoria de la aplicación.

Consecuencias directas de estas características de implementación son:

- Todo modelo ha de contener una referencia navegable a su meta-modelo. Si el meta-modelo no existe o no está accesible, el modelo no puede ser cargado en el entorno. El meta-meta-modelo EMOF se encuentra implementado directamente en el código del entorno, justamente porque es meta-modelo de sí mismo y no puede cumplir esta restricción.
- Creado o cargado un meta-modelo, se puede crear programáticamente cualquier modelo conforme a él, sin requerir pre-procesamiento previo del meta-modelo ni generación del código fuente de sus clases.
- Puesto que es muy severa la restricción de que un modelo referenciado desde otro modelo deba estar cargado en el entorno antes que él, PyEmofUC permite mantener referencias a elementos de modelos externos mediante URIs textuales. Cuando

esto ocurre, la referencia es completa, aunque los programas no pueden acceder al elemento referenciado por ella.

- La implementación Python de un modelo sólo existe en el entorno en que ha sido creado. Su almacenamiento persistente en el sistema de ficheros o en servidores *web* y su intercambio con otros entornos, requiere una conversión previa a un formato textual XMI. El entorno proporciona recursos para importar y exportar modelos a y desde los otros espacios tecnológicos.

Interfaz pública del entorno. En la Fig. 4 se muestran los elementos que constituyen la interfaz pública para la gestión de los modelos que ofrece el entorno a las aplicaciones. La interfaz está constituida por las propiedades y métodos de las clases del entorno:

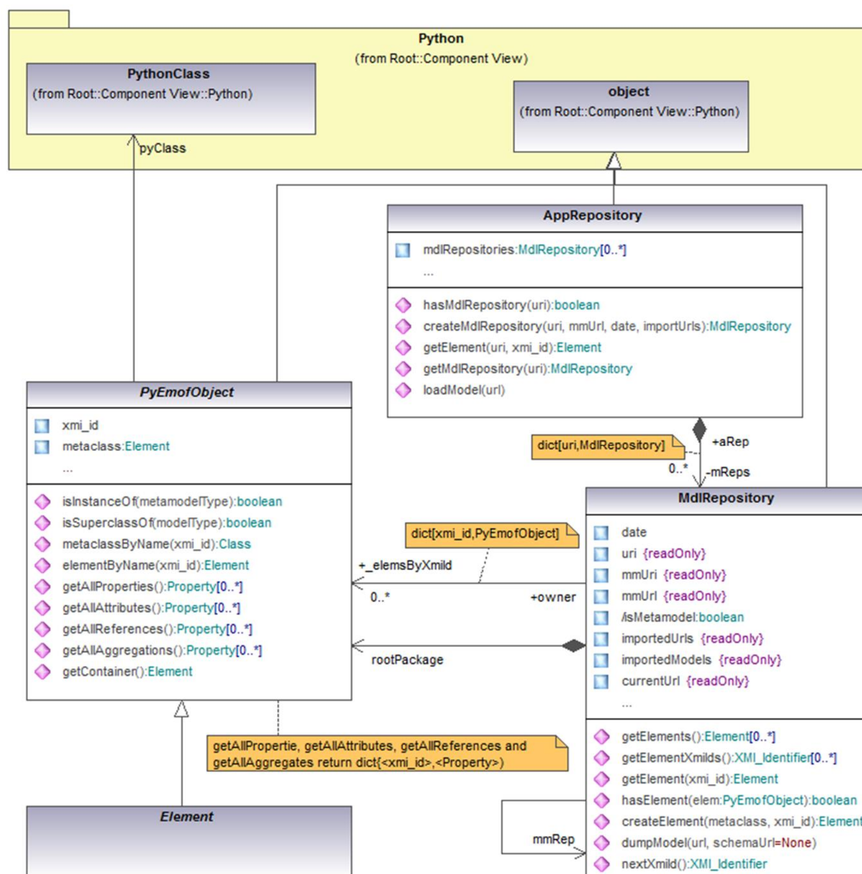

Fig. 4. Interfaz pública del entorno PyEmofUC.

- La clase *AppRepository* describe los repositorios que contienen los modelos en el espacio de memoria de las aplicaciones. Una aplicación puede definir múltiples

instancias de esta clase, y cada una de ellas, puede contener múltiples modelos que son accesibles a la aplicación. Las aplicaciones hacen uso de las propiedades y métodos de esta clase para la gestión de los modelos como entes:

  - El método *createMdlRepository()* crea un nuevo modelo vacío en el repositorio, mientras que el método *loadModel()* crea un modelo completo a partir de una formulación serializada del mismo.
  - La propiedad *mdlRepositories* permite acceder al diccionario del repositorio que contiene los modelos creados en el repositorio indexados por su URI, y el método *getMdlRepository()* proporciona la referencia a un modelo con el URI especificado como parámetro.
  - La operación *getElement()* proporciona la referencia de un elemento de un modelo en base al URI del modelo y al identificador del elemento dentro de él.

- La clase *MdlRepository* describe los repositorios que contienen los elementos de un modelo. A través de su interfaz pública, las aplicaciones acceden a las propiedades globales del modelo y gestionan los elementos que contiene:
  - La propiedad *aRep* referencia el repositorio de aplicación en el que ha sido creado el modelo.
  - Las propiedades *date*, *uri*, *nsPrefix* e *isMetamodel* contienen características globales del modelo.
  - Las propiedades *mmUri, mmUrl* y *mmRep* representan respectivamente el URI, el localizador y la referencia relativos a su meta-modelo.
  - Las propiedades *importedUrls* e *importedModels* contienen los localizadores y las referencias a otros modelos referenciados por el modelo.
  - La propiedad *rootPackage* referencia el elemento contenedor raíz de los elementos del modelo definidos en su descripción EMOF.
  - La propiedad *currentUrl* es el localizador de una copia actualizada y almacenada persistentemente del modelo. Cuando el modelo se crea o se cambia, esta propiedad se establece a *None*. Cuando el modelo se carga (*loadModel()*) o se almacena (*dump()*), se establece al correspondiente localizador.
  - Los métodos *getElements()*, *getElement()*, *getElementXmiIds()* y *hasElement()* son diferentes formas de acceso a los elementos del modelo.
  - El método *dump()* genera una formulación serializada del modelo, que la aplicación puede utilizar para almacenar persistentemente el modelo o para intercambiarlo con otro entorno.
- La clase *PyEmofObject* es la clase raíz de todos los elementos gestionado por el entorno. Define los atributos que el entorno PyEmofUC requiere para la gestión de cualquier elemento. Así mismo, define el conjunto de los métodos reflexivos que, por herencia, se aplican a todos los elementos del modelo.
  - La propiedad *xmi_id* es el identificador único de un elemento dentro del modelo. Su nombre es reflejo de que su valor también se utiliza en la formalización XMI del modelo.
  - La propiedad *metaclass* referencia el elemento del meta-modelo que describe su estructura de datos (propiedades) y su comportamiento (operaciones). Es la base de la implementación de la reflexividad en el entorno.

– La propiedad *pyClass* sólo está establecida en los elementos *DataType* de los meta-modelos y referencia la clase Python con la que se crean las instancias de tales tipos de datos en el nivel inferior.

- La información propia del modelo, tal y como se describe en su meta-modelo, se define a través de los elementos especializados de la clase EMOF::*Element* y de los valores que se asignan a sus propiedades. Así mismo, el comportamiento de los elementos del modelo se define a través de las operaciones que se declaran en sus meta-clases. En la implementación Python a todos los elementos (clases, propiedades, operaciones y parámetros) se le asignan los nombres definidos en su descripción EMOF.

EmofUC. En el entorno PyEmofUC se utiliza el meta-meta-modelo *EmofUC*, que es una extensión de la especificación EMOF 2.4 de OMG. En él se han introducido dos leves modificaciones para simplificar su implementación Python. Se han eliminado algunas de las clases abstractas incluidas en EMOF como consecuencia de su especificación en base a UML2 y CMOF, pero que son irrelevantes en EMOF. Así mismo, se ha cambiado el nombre de dos referencias (EMOF::type →EmofUC::theType y EMOF::class →EmofUC::theClass) cuyos nombres originales son palabra reservadas en el lenguaje Python.

En *EmofUC* se ha especificado el conjunto de tipos primitivos, que se define en el correspondiente anexo de UML2. Éstos se han renombrado como *PyString*, *PyBoolean*, *PyReal*, *PyInteger* y *PyUnlimitedInteger*, ya que tienen la semántica de UML pero asumen los rangos de valores propios de los tipos primitivos *str*, *bool*, *float*, *int* y *long* de Python. Así mismo, se han definido otros tipos de datos primitivos (*PyToken*, *XMI_Identifier*, *URI*, *URL*, *DefaultValue*, *NsPrefix*, *LowerCardinality*, *UpperCardinality*, *GlobalPropertyRef*, *GlobalClassRef* y *GlobalTypeRef*) a fin de eliminar laxitud en los tipos de las propiedades de los meta-modelos.

Utilizando los elementos *tags* definidos en EMOF, se ha introducido un conjunto de extensiones en *EmofUC* que incrementa su capacidad respecto a EMOF:

- Los *tags* de nombre reservado "PT" permiten definir herencia entre tipos de datos, mientras que los *tags* de nombres reservados "RE", "MAX" y "MIN" permiten definir los rangos de valores del tipo. Por ejemplo, en la Tabla 1 se muestra la definición (en formato XMI) de un tipo primitivo *Percentage* que es un subtipo del tipo primitivo *PyReal* y con rangos limitados entre 0.0 y 100.0.

Table 1. Definición del tipo primitivo Percentage.

```
<ownedType  name="Percentage" xmi:id="mast2.Percentage" xsi:type="emof:PrimitiveType">
    <tag name="PT" xmi:id="mast2.Percentage.PT" xsi:type="emof:Tag" value="emof#emof.PyReal"/>
    <tag name="MAX" xmi:id="mast2.URI.MAX" xsi:type="emof:Tag" value="100.0"/>
    <tag name="MIN" xmi:id="mast2.URI.MIN" xsi:type="emof:Tag" value="0.0"/>
</ownedType>
```

- Los *tags* de nombre reservado “DPE” permiten especificar el código con el que se evalúa una propiedad derivada. En la Tabla 2 se muestra la declaración de la propiedad derivada *isRoot()* que EMOF define para los elementos *Package*. La declaración de la Tabla 2 equivale a incluir (en tiempo de ejecución) en la implementación Python de las clase Package, la declaración de una propiedad definida por el código:

```
def getisRoot(self): return self.nestingPackage==None
isRoot= property(getisRoot)
```

Table 2. Declaración de la propiedad derivada *isRoot*.

```
<ownedType name="Package" xsi:type="emof:Class" xmi:id="emof.Package" superclass="#emof.NamedElement">
   ........
    <ownedAttribute name="isRoot" xmi:id="emof.Package.isRoot" xsi:type="emof:Property"
                  theType="#emof.PyBoolean" isDerived="True" >
         <tag name="DPE" xmi:id="emof.Package.isRoot.DPE" xsi:type="emof:Tag"
                     value="return self.nestingPackage==None"/>
    </ownedAttribute>
</ownedType>
```

- Haciendo uso del *tag* con nombre reservado “OPE”, se puede asociar el código que implementa las operaciones definidas en las clases de los meta-modelos y que el entorno genera automáticamente en las correspondientes clases. La extensión es útil para incorporar el código sencillo de las operaciones de gestión de los objetos del modelo.

3.2 Representación textual de los modelos.

Los modelos se formalizan como un texto con un formato estándar independiente de los lenguajes de programación y de las plataformas de ejecución, cuando se transfieren de un entorno a otro, se almacenan aisladamente o son mostrados a un operador. En esta formulación, las referencias entre elementos del mismo o de diferentes modelos se formulan mediante URIs y los localizadores de los modelos como URLs.

En la Fig. 5 se muestran los dos formatos textuales que se definen en el entorno PyEmofUC:

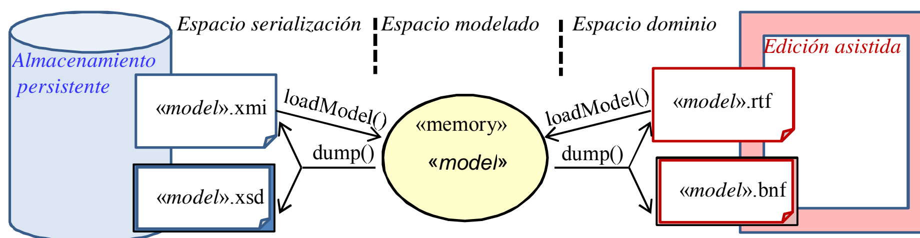

Fig. 5. Opciones de serialización.

- *«model».xmi*: Es un formato de texto etiquetado, formulado de acuerdo con el estándar XMI de OMG [14]. Se utiliza tanto para el almacenamiento de la información en el sistema de ficheros de las plataformas de ejecución o en servidores remotos como para el intercambio con otros entornos a través de flujos de bytes (*streaming*).
- *«model».rtf*: Se formulan de acuerdo con la gramática formal definida en PyEmof UC y se codifica con el formato estándar RTF para enriquecerlo con color. Se utiliza para la presentación de los modelos a expertos de dominio haciendo uso de editores asistidos.

Tanto el esquema *«model».xsd* como la gramática formal *«model».bnf* del lenguaje textual de dominio son generados automáticamente por el entorno cuando se serializa el correspondiente meta-modelo a formato XMI usando el método *dump()*. Las descripciones de los dos formatos pueden encontrarse en la referencia [1].

4 Transformaciones entre modelos.

Las transformaciones entre modelos se formulan como funciones Python que procesan la información en el espacio de modelado. La fiel correspondencia entre la estructura de datos Python en este espacio y la representación EMOF de los modelos hace que cualquier experto de dominio que conozca éstos, pueda acceder de forma sencilla a la información de un modelo cargado en el entorno y abordar la creación de nuevos modelos. Al ser Phyton un lenguaje que admite los estilos de programación procedural, orientado a objetos y funcional, permite formular las transformaciones de modelos con estilo imperativo, declarativo o híbrido (combinación de los dos anteriores).

Cuando se utiliza un estilo imperativo, la transformación de un modelo se realiza a través de bucles y sentencias condicionales que exploran el modelo de entrada para generar, en base a los elementos encontrados, los elementos del modelo de salida. En PyEmofUC, esto se realiza con una función Python utilizando su estilo de programación procedural u orientado a objetos.

Cuando se utiliza un estilo declarativo, la transformación de un modelo se especifica a través de una regla para cada tipo de elemento del modelo de entrada que define el elemento del modelo de salida que debe generarse, así como los valores de los atributos y referencias que se le ha de asignar en la transformación. La transformación la realiza un motor de transformación que itera sobre los elementos especificados. En PyEmofUC, esto se realiza en dos fases: en la primera, se generan los elementos del modelo de salida utilizando un estilo de programación funcional, sentencias de compresión de listas y generadores, y en la segunda fase, se resuelven las referencias Python entre los elementos del modelo de salida generados en la primera fase.

Una técnica importante dentro de MDE es la posibilidad de representar una transformación como un modelo que se genera por transformación de otros modelos. Aunque en la versión actual no está aún implementado, está previsto que el núcleo del entorno ofrezca una operación que ejecute un modelo formulado de acuerdo con la

especificación MOF QVT Relations [15] de OMG. Con ella, las transformaciones M2M generadas por transformación de modelos podrán ser ejecutadas.

5 Procesos básicos en el entorno PyEmofUC

En esta sección se muestran ejemplos de las operaciones básicas de gestión de los modelos en el entorno PyEmofUC.

En esta sección se utiliza un caso de uso representativo del entorno. Consiste en la formulación de un comando Python que realice el análisis de planificabilidad de modelos en el entorno MAST [16] cuando coexisten diferentes versiones. En este momento, se ha introducido una revisión mayor en MAST. Los modelos se formulan según la nueva versión MAST 2.0, pero muchas de las herramientas del entorno sólo operan aún con la versión antigua MAST 1.4. La Fig. 6 describe el proceso que sigue el comando *analyses_script.*

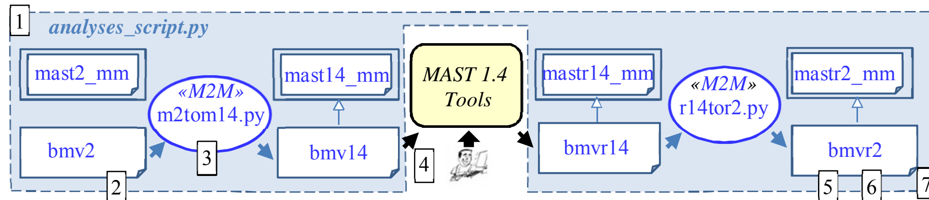

Fig. 6. Caso de uso: Análisis del modelo MAST2:bmv2 con herramientas MAST 1.4.

La entrada al proceso es un modelo (p.e.õ*bmv2*ö) formulado en la actual versión MAST 2.0, esto es, conforme al meta-modelo "*http://unican.es/istr/MAST2/MastModel*" (en la figura "*mast2_mm*") y la salida es el modelo de resultados (p.e.õ*bmvr2*ö) conforme al meta-modelo "*http://unican.es/istr/MAST2/MastResult*" (en la figura "*mastr2_mm*"). Para ser compatible con la herramienta MAST 1.4, debe ser transformado a modelo conforme con el meta-modelo MAST 1.4, y los resultados transformados a la versión final. Los ejemplos de operaciones básicas que a continuación se exponen son segmentos de este comando.

1. *Creación y acceso al entorno PyEmofUC*. Una aplicación que va a operar sobre el entorno PyEmofUC debe crear previamente un repositorio de aplicación. Desde su interfaz pública se cargan, transforman y visualizan los modelos. La creación del repositorio se realiza invocando al constructor *AppRepository()*.

```
>>> import emofUC.resource as resource
>>> aRep=resource.AppRepository()     # aRep es la referencia al repositorio de aplicación sobre el que se opera
>>>
```

2. *Carga de un modelo desde un fichero*. La carga de un modelo en el entorno se realiza invocando el método *loadModel(aRep, modelURL)* en el repositorio de la aplicación, lo cual implica obligatoriamente la carga de su meta-modelo y de cualesquiera otros modelos referenciados por él (en este ejemplo ninguno).

```
>>> bmv2= aRep.loadModel('file:///c:/PyEmof_WS/BMV/bmv2.xmi') # bmv2 es la referencia al modelo cargado
>>>
```

La carga del modelo *bmv2* ha realizado también la carga de su meta-modelo *http://unican.es/istr/MAST2/MastModel*, y al ser el primero, también del meta-meta-modelo EmofUC.

Los modelos que están cargados en el entorno puede visualizarse a través del método *getMdlRepositories()*, que retorna un diccionario con todos los modelos cargados en el entorno e indexados por su respectivo URI.

```
>>> for key in aRep.getMdlRepositories().keys(): print "- "+key
    - http://unican.es/istr/PyEmofUC/BMV/bmv2
    - http://unican.es/istr/PyEmofUC/EmofUC
    - http://unican.es/istr/MAST2/MastModel
>>>
```

3. *Transformación de un modelo en otro modelo*. La transformación de un modelo se realiza invocando la función Python que la implementa. En este caso corresponde al método *m2tom14()* del módulo *mast2ToMast14Scripts.py*. Este método crea en el entorno un modelo transformado con el URI que se pasa como parámetro.

```
>>> import mast2.mast2ToMast14Scripts as script
>>> script.m2tom14(bmv2,' http://unican.es/istr/PyEmofUC/BMV/bmv14)
>>>
```

4. *Intercambio de modelos con otro entorno*. MAST es un conjunto de herramientas implementadas en Ada, y por tanto constituye otro entorno que no es de tipo PyEmofUC. La transferencia del modelo *bmv14* generado por el comando al entorno Ada-MAST ha de realizase a través de serialización de los modelos. El formato de intercambio XMI es común a ambos. En este caso, la información se serializa haciendo uso del método *dump()* y el modelo de resultados que se retorna se carga haciendo uso del método *loadModel()*. El que la transferencia se haga a través de un *socket* (MAST_HOST, MAST_PORT) o se haga a través de ficheros, es irrelevante al objetivo que persigue el ejemplo.

```
>>> MAST_HOST = '...'                                    # Creates socket bound wirt MAST server
>>> MAST_PORT = 39000
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> s.connect((MAST_HOST, MAST_PORT))
>>> bmv14=aRep.getMdlRepository('http://unican.es/istr/PyEmofUC/BMV/bmv14') #Get reference to bmv14 model
>>> bmv14XMI=model.dump(url=None,schemaUrl=None)  # Sequentializes the model as XMI
>>> s.sendall(modelXMI)                                  # Send the sequentialized model
>>> bmvr14XMI= s.recv(INPUT_BUFF_LEN))                    # Receives the results as XMI
>>> bmvr14Model=aRep.loadModel(resultXMI)                # Loads the result model in aRep repository
>>> s.close()                                            # Closes the socket
>>>
```

5. *Visualización de la información de un modelo*. Un modelo puede ser visualizado por el operador en dos formatos: como un documento XML, forma nativa que se visualiza a través de un editor XML con la asistencia para la edición del correspondiente W3C-schema generado por el entorno, o como un texto plano que sigue la gramática EBNF también generada por el entorno y que se visualiza a través de un editor de texto enriquecido con color y que soporta el formato RTF. En la versión actual, la visualización se realiza invocando el editor correspondiente sobre el fichero que almacena el modelo en el formato deseado. Éste debe ser almacenado previamente haciendo uso del método *dump()* aplicado a una instancia del modelo disponible en el entorno.

```
>>> bmvr2= aRep.getMdlRepository('http://unican.es/istr/PyEmofUC/BMV/bmvr2')
>>> bmvr2.dump(url='file:///c:/pyEmof_WS/BMV/bmvr2.rtf")
>>>
```

6. *Almacenamiento persistente de un modelo*. La creación, transformación y procesamiento de los modelos se realiza en memoria de las aplicaciones y éstos permanecen en ella mientras el repositorio de aplicación no se destruya. Cuando se quiere salvar persistentemente un modelo, éste ha de convertirse a formato XMI, utilizando el método *dump(url)* sobre él, y especificando un localizador con la extensión *õ.xmiö*.

```
>>> bmvr2= aRep.getMdlRepository('http://unican.es/istr/PyEmofUC/BMV/bmvr2)
>>> bmvr2.dump(url='file:///c:/PyEmof_WS/BMV/bmvr2.xmi', schemaUrl= http://unican.es/istr/mast2/MastResult.xsd)
>>>
```

7. *Eliminación de los modelos del entorno*. En la versión actual de PyEmofUC no se admite la eliminación individual de un modelo del repositorio. Los modelos en memoria se eliminan cuando desde la aplicación se elimina el repositorio de aplicación que lo contiene.

```
>>> del aRep
>>>
```

6 Conclusiones y trabajo futuro

La principal característica del entorno PyEmofUC es su minimalismo, esto es, la unión de la ligereza de su núcleo, la sencillez de su interfaz pública y la no necesidad de infraestructuras para el desarrollo de aplicaciones MDE. Para conseguirlo, se requiere que cada modelo o meta-modelo incluya individualmente los localizadores de los otros modelos o de los recursos que se necesitan para su interpretación. En otros entornos MDE, el manejo y diferenciación entre URI y URL es más suave, ya que con el uso de espacios de trabajo la información es registrada como parte del entorno.

En la versión actual del entorno se proporcionan algunas operaciones y pequeñas aplicaciones de soporte externas a su núcleo, como verificadores, editores asistidos, etc. En esta línea, conviene avanzar para que el entorno sea más robusto y amigable.

Un aspecto relativo al núcleo que sí se va a desarrollar próximamente es la incorporación de un cuarto espacio tecnológico relativo al soporte de los modelos en servidores de bases de datos. Con él, se tendrá la capacidad de gestionar modelos que por la cantidad de información que contienen no pueden ser mantenidos en la memoria de la aplicación como estructura de datos Python.

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

References

1. *Web de soporte del entorno PyEmofUC. http://www.istr.unican.es/pyemofuc/*
2. "formal/2014-04-03: Meta Object Facility (MOF) Core, v2.4.2," 2014.
3. *õUniform Resource Identifier (URI): Generic Syntaxö Network Working Group http://www.ietf.org/rfc/rfc3986.txt*.
4. *PyEMOF: http://www.lifl.fr/~marvie/software/tutorial.html*
5. *Eclipse Modeling Project: http://www.eclipse.org/modeling*
6. R. C. Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Addison-Wesley Professional, 2009.
7. *Microsoft DSL Tools. https://msdn.microsoft.com/en-us/library/bb126327.aspx*
8. S. Cook, G. Jones, S. Kent and A. C. Wills, *Domain-Specific Development with Visual Studio Dsl Tools.* Pearson Education, 2007.
9. IBM Rational Software Architect: http://www.ibm.com/developerworks/downloads/r/architect
10. *The Generic Modeling Environment:http://www.isis.vanderbilt.edu/projects/gme*
11. J. Davis, "GME: The generic modeling environment," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications,* 2003, pp. 82-83.
12. *FUJABA tool suite: http://www.fujaba.de*
13. *Obeo Designer: http://www.obeodesigner.com*
14. "formal/2014-04-04: XML Metadata Interchange (XMI), v2.4.1," 2014.
15. "formal/2015-02-01 (Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.2)," 2015.
16. M. G. Harbour, J. J. Gutiérrez, J. L. Medina, J. C. Palencia, J. M. Drake, J. M. Rivas, P. L. 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 Burnsø Sixtieth Birthday,* pp. 42.