Maude como Soporte Formal para una Herramienta de Gestión de Modelos♦

[1]Francisco J. Lucas, [2]Artur Boronat, [1]José L. Fernández, [2]José Á. Carsí, [1]Ambrosio Toval, [2]Isidro Ramos

[1]Departmento de Informática y Sistemas
Universidad de Murcia
Campus de Espinardo
30071 Murcia
{fjlucas | aleman | atoval}@um.es

[2]Departmento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
C/Camí de Vera s/n
46022 Valencia
{aboronat | pcarsi | iramos }@dsic.upv.es

Resumen

La gestión de modelos engloba todo tipo de tareas que permitan su representación y/o manipulación, entre ellas, la evolución de sistemas software, migración de datos, la interoperabilidad, etc. Este trabajo se centra en el análisis de la teoría algebraica de la plataforma para la gestión de modelos MOMENT [1] y el posible uso de la reflexión del lenguaje formal *Maude* [2], para mejorar las características de esta especificación. Una álgebra para MOMENT ha sido expresada en *CafeOBJ* y presenta una serie de *sorts* que constituyen la estructura de la plataforma en diferentes niveles de abstracción y que forman la meta-arquitectura de la plataforma. Esta estructura permite abordar la construcción de metamodelos, modelos e instanciaciones, aportando las operaciones necesarias para definirlos y manipularlos. La adaptación de esta teoría al lenguaje *Maude* y el uso de la reflexión de este lenguaje permitirán tener otro nivel más de abstracción por encima de los que defina la teoría que nos otorgará un nuevo grado de libertad al poder manipular la propia plataforma MOMENT.

Palabras Clave: Maude, lógica de reescritura, reflexión, gestión de modelos, metamodelado.

1 Introducción

El desarrollo de software implica el estudio del sistema de información que constituye el problema a resolver (espacio del problema), las tecnologías que permiten implementar una solución (espacio de la solución) y el proceso de desarrollo del software que obtiene una solución tecnológica a los requisitos del sistema de información.

Dicho método organiza una serie de herramientas que permiten describir el producto software a desarrollar con diferente grado de abstracción: requisitos, diseño de la aplicación, tanto de la semántica como de la persistencia, arquitectura y modelos específicos de plataforma que tienen en cuenta una tecnología concreta para la implementación final del producto software.

Todos estos artefactos software pueden actuar como modelos con un nivel específico de abstracción. Tal y cómo apunta la iniciativa Model Driven Architecture [5] de OMG mediante la propuesta de estándar Query/View/transformations [8], el proceso de desarrollo de software se puede ver como una sucesión de transformaciones de modelos, desde un elevado nivel de abstracción, como los requisitos,

♦ Este artículo ha sido financiado por el Proyecto Nacional de Investigación, Desarrollo e Innovación DYNAMICA, subproyectos PRESSURE TIC 2003-07804-C05-05 y PRISMA TIC 2003-07804-C05-01.

hasta un nivel específico, como lo es el código que implementa la aplicación. La formalización de los modelos y de las transformaciones entre ellos facilita la automatización del proceso de desarrollo de software. Las técnicas formales también permiten aplicar mecanismos de validación sobre los modelos garantizando un software de calidad como resultado del proceso de desarrollo de software.

Siguiendo este enfoque ha surgido un campo de investigación que pretende dar solución a este tipo de problemas: la gestión de modelos. Un modelo es una representación de un dominio que permite la comunicación entre los diseñadores que participan el en desarrollo de la aplicación. Siguiendo esta aproximación, en la Ingeniería del Software, un modelo es una estructura que representa un artefacto de diseño, como un esquema relacional, un diagrama UML, un DTD XML, etc. La gestión de modelos persigue dar solución a la representación de modelos y a su manipulación de un forma automática y genérica.

MOMENT [3] es una herramienta que permite definir modelos como especificaciones algebraicas de una teoría expresada en lógica ecuacional condicional. MOMENT proporciona un mecanismo para transformar modelos basados en el álgebra que utiliza la herramienta, basándose en el soporte formal y automático que proporcionan los sistemas de reescritura de términos [4], como Maude, CafeOBJ, etc. Este mecanismo de transformación permite automatizar gran cantidad de procesos software, como por ejemplo, los estudiados en el contexto Model-Driven Architecture (MDA) [5], que permiten entre otras cosas generar modelos específicos de plataforma partiendo de modelos independientes de plataforma.

El lenguaje Maude es un lenguaje de especificación algebraica basado en lógica ecuacional y lógica de reescritura. En su desarrollo sus autores han intentado potenciar la expresividad, la sencillez y el rendimiento del lenguaje. Además de estas propiedades, Maude tiene otras cualidades que le hacen único dentro del campo de los lenguajes algebraicos, que son, entre otras: la posibilidad de ejecutar las especificaciones, soporte para programación OO, permite la creación de módulos parametrizados, la posibilidad del uso de la reflexión. En este trabajo se hará uso de esta última cualidad para proporcionar mayor flexibilidad a la teoría algebraica de MOMENT, permitiendo manipular con más libertad los modelos que se definan así como la propia especificación algebraica.

Tras esta introducción, la Sección 2 dará una visión general de la teoría algebraica usada en MOMENT. En la Sección 3, se verán de forma más extensa las características del lenguaje formal Maude, así como un pequeño estudio del uso de la reflexión en la teoría MOMENT. Por último, presentaremos las conclusiones y las posibles vías futuras de investigación como continuación a este trabajo.

2 La Teoría Algebraica de MOMENT

En este apartado se describen las características generales de la teoría algebraica usada en la plataforma de gestión de modelos MOMENT, y se describen los sorts que participan en ella. Esta teoría ha sido expresada en *CafeOBJ* [1] y permite la representación y manipulación de modelos en la plataforma MOMENT, sobre el sistema de reescritura *CafeOBJ* [6]. También se comenta alguno de los inconvenientes que tiene el álgebra actual de MOMENT y que pueden ser solucionados mediante las capacidades reflexivas de *Maude*.

2.1 Características Principales

Mediante la teoría algebraica de MOMENT se ha especificado la estructura de la plataforma, constituida por cuatro niveles de abstracción, siguiendo la cultura de metamodelado presentada en el estándar MOF [7], aunque no implementa directamente su lenguaje abstracto para definir metamodelos. Cada nivel está formado por un conjunto de esquemas que a su vez están constituidos por un conjunto de conceptos y propiedades. Los conceptos permiten representar entidades descriptibles en un dominio determinado, y las propiedades establecen dichas descripciones, pudiendo asociar dos conceptos mediante una propiedad o bien describir un concepto mediante una propiedad que contiene un valor básico. Los niveles que consituyen la plataforma MOMENT son los siguientes:

– Nivel M3: contiene los elementos (conceptos y propiedades) necesarios para poder describir cualquier metamodelo en el siguiente nivel. Es el nivel más abstracto de la plataforma.

- Nivel M2: sus esquemas constituyen metamodelos, donde sus respectivos conceptos y propiedades indican como definir un modelo en el siguiente nivel.
- Nivel M1: sus esquemas representan modelos de un determinado metamodelo del nivel M2. Los conceptos y propiedades que forman un modelo permiten definir información en el siguiente nivel.
- Nivel M0: sus esquemas representan instanciaciones de modelos del nivel M1 que contienen datos concretos.

La estructura utilizada para representar modelos está basada en la "tripleta RDF" que está formada por un sujeto (*domain*), un predicado y un objetivo (*range*), y donde se puede especificar propiedades de objetos para un elemento sujeto específico mediante relaciones de predicado, pudiendo utilizar un sujeto como objeto de otra propiedad. De esta forma, se puede representar cualquier artefacto software en forma de grafo RDF.

2.2 Elementos del Álgebra

Esta álgebra consiste en una serie de *sorts* y un conjunto de operaciones definidos sobre ellos. El lenguaje de especificación elegido para representar los elementos del álgebra MOMENT es *CafeOBJ*, que está basado en lógica ecuacional y de reescritura. Los elementos más importantes que forman esta álgebra son:

- *Layer*, representa un nivel de abstracción. Y contiene un conjunto de artefactos llamados esquemas. El constructor de este elemento sería:

  op layer _ _ : LayerId ListSchema -> Layer {constr}

  Existen cuatro términos de este *sort*, uno por cada nivel de la plataforma.
- *Schema*, representa un conjunto de conceptos y propiedades que describen esos conceptos. Su constructor es el siguiente:

  op schema _ _ _ _ : LayerId SchemaId SchemaId ListNode -> Schema {constr}

  Donde cada término de este *sort* es una tupla *s = (layer_id, schema_parent_id, schema_id, list_nodes)*, donde: *layer_id* es el identificador del nivel al que pertenece el esquema, *schema_parent_id* es el identificador del esquema padre de este esquema, *schema_id* es el identificador del esquema que se está definiendo, *list_nodes* es un término del sort *ListNode* constituyendo una lista de nodos, que pueden ser conceptos o propiedades, como veremos más adelante, y que forma el cuerpo del esquema.
- *Concept*, representa a un recurso RDF. El constructor de este *sort* sería el siguiente:

  op concept _ _ _ _ : LayerId SchemaId ConceptId ConceptId -> concept {constr}

  Donde un término se puede ver como una tupla *c = (layer_id, schema_id, parent_id, id)*, donde: *schema_id* es el identificador del esquema que contiene el concepto, *layer_id* es el identificador del nivel que contiene dicho esquema, *parent_id* es el identificador del metaconcepto situado en un esquema del nivel inmediatamente superior al identificado por layer_id, y que representa el tipo del concepto que está, *id* es el identificador URI del concepto.
- *Property*, sirve para especificar propiedades. Existen tres tipos de elementos que pueden participar en una proposición RDF: conceptos, tipos de valor (*Datatype*), y valores (*Value*). Según el tipo de elemento objeto que participa en la propiedad identificamos tres tipos de propiedades:
  - *Concept property*, que es una propiedad cuyo rango es un concepto. El constructor de este *sort* sería el siguiente:

    op conceptProperty _ _ _ _ _ _ _ _ :
    LayerId SchemaId PropertyId PropertyId Cardinality
    Cardinality NodeId ConceptId -> Property {constr}

Donde un término de este *sort* tendría esta forma: *p = (layer_id, schema_id, parent_id, id, min_card, max_card, domain_id, range_id)*, donde: *layer_id* es el identificador del nivel que contiene el esquema identificado por *schema_id*, *schema_id* es el identificador del esquema que contiene la propiedad, *parent_id* es el identificador de la metapropiedad que representa el tipo de la propiedad que está situada en un esquema del nivel inmediatamente superior al identificado por *layer_id*, *id* es el identificador URI de la propiedad, *min_card* es la cardinalidad mínima de la propiedad e indica la cantidad mínima de instancias del concepto rango, que pueden estar relacionadas con una instancia del nodo sujeto de la propiedad, *max_card* es la cardinalidad máxima de la propiedad e indica la cantidad máxima de instancias del concepto rango que se pueden relacionar con una instancia del nodo sujeto, *domain_id* es el identificador del nodo que actúa como sujeto de la propiedad, *range_id* es el identificador del concepto que actúa como objeto de la propiedad.

- *Datatype property*, define un tipo de datos del valor objeto. Su constructor es el siguiente:

  op datatypeProperty _ _ _ _ _ _ _ _ :
  LayerId SchemaId PropertyId PropertyId Cardinality
  Cardinality NodeId Datatype -> Property {constr}

  Un término de este *sort* tendría la forma de esta tupla *p = (layer_id, schema_id, parent_id, id, min_card, max_card, domain_id, datatype)*, donde *datatype* representa uno de los tipos básicos soportados en la plataforma.
- *Value property*, representa un objeto de una propiedad que es un valor básico. Su constructor sería el siguiente:

  op valueProperty _ _ _ _ _ _ _ _ :
  LayerId SchemaId PropertyId PropertyId Cardinality
  Cardinality NodeId Value -> Property {constr}

  La tupla que representaría un término de este sort sería: *p = (layer_id, schema_id, parent_id, id, min_card, max_card, domain_id, value)*, donde *value* constituye un valor básico de un tipo específico.

– *Node*, representa un concepto o una propiedad.

2.3 Manipulación de Modelos

En este álgebra se define un conjunto de operadores que permiten manipular (crear, eliminar y actualizar) los elementos del álgebra respetando la jerarquía de cuatro niveles de abstracción. Para ello se definen para cada operación y cada *sort* las precondiciones que garanticen que la operación no dejará inconsistente el sistema.

A continuación se presenta un ejemplo del tipo de precondiciones que se establecen sobre un operador con la operación de eliminación de un *Schema*, que se hace a través de la operación *safeDestroy*. Para poder borrar un esquema se deben satisfacer las siguientes precondiciones:

1. El esquema a ser borrado S no puede estar instanciado en un nivel inferior. Así evitamos tener que borrar las instancias S' del esquema S en el nivel inferior, y el consecuente borrado de los esquemas S'' que sean instancias de S' en el siguiente nivel inferior, y así recurrentemente. Se obliga a eliminar todas las instancias de S para permitir su borrado.
2. El esquema que va a ser borrado debe estar definido en el nivel indicado.

Este tipo de precondiciones limitan mucho la libertad a la hora de manipular el modelo. En el siguiente apartado se indica como el mecanismo de reflexión de *Maude* permite solventar este inconveniente.

3 El lenguaje Formal *Maude*

En este apartado veremos las características más interesantes del lenguaje *Maude* y describiremos brevemente la forma de usar la reflexión en *Maude*, y su aplicación en el álgebra MOMENT.

3.1 Características principales de Maude

Maude es un lenguaje de especificación formal, basado en lógica ecuacional y lógica de reescritura. En *Core Maude* existen tres tipos de módulos (funcionales, de sistema y OO, aunque estos últimos no ofrecen una verdadera programación OO). Además, *Maude* ofrece una versión "ampliada", llamada *Full Maude*, realizada sobre *Maude* y que da la opción de utilizar una verdadera programación OO, y de utilizar módulos parametrizados, ésta es una de las funcionalidades más potentes de *Maude*.

Además, *Maude* es un lenguaje que permite la ejecución de las especificaciones que hemos creado, lo que permitirá animar los modelos y crear prototipos del sistema para comprobar el funcionamiento del sistema.

Otra de las características de *Maude*, y que lo diferencia de otros lenguajes algebraicos, es la posibilidad de usar reflexión. En los siguientes apartados veremos como funciona la reflexión en *Maude*, y como puede ser utilizada para mejorar el álgebra de la plataforma MOMENT.

3.2 La Reflexión en Maude

La lógica de reescritura es una lógica reflexiva, esto es, que puede ser interpretada en si misma fielmente. El diseño de *Maude* hace posible el uso de reflexión en su lógica de reescritura.

Esta reflexión de la lógica de reescritura está definida de forma precisa desde un punto de vista matemático. Existe una teoría de reescritura *U* que es "universal", de forma que cualquier teoría de reescritura *R* (incluida *U*) puede ser representada en *U* como un término $\overline{R}$. Cualquier termino *t* y *t'* definido en *R* son términos $\bar{t}$ y $\bar{t}'$ en $\overline{R}$, y cualquier par (*R*, *t*) es un término $\langle \overline{R}, \bar{t} \rangle$, de forma que se cumple la siguiente equivalencia:

$$R \mapsto t \to t' \Leftrightarrow U \mapsto \langle \overline{R}, \bar{t} \rangle \to \langle \overline{R}, \bar{t}' \rangle$$

Como *U* es representable por sí misma, se logra una "torre de reflexión" que permite tener un número arbitrario de niveles de reflexión.

En *Maude*, esta teoría universal *U* ha sido implementada eficientemente a través del módulo *META-LEVEL*. Este módulo incluye a su vez otros dos que son: *META-MODULE* y *META-TERM*. Concretamente:

- en el módulo *META-TERM*, los términos *Maude* son meta-representados como elementos del *sort Term*, que es el tipo de datos de los términos,
- en el módulo *META-MODULE*, los módulos *Maude* son meta-representados como elementos del *sort Module*, que es el tipo de datos de los módulos, y
- en el módulo *META-LEVEL*, que implementa la meta-representación de los comandos *Maude*: *reduce*, *apply*, …, a través de las funciones *metaReduce*, *metaApply*,… Estas funciones son llamadas "*descent funtions*", ya que permiten descender en la torre de reflexión.

Vamos a ver con un poco más de detalle el funcionamiento del módulo *META-MODULE*, que será el que se utilice para modificar el álgebra MOMENT. En este módulo, que importa el módulo *META-TERM*, los módulos funcionales y de sistema que contiene *Maude* son meta-representados con una sintaxis muy similar a su sintaxis original. La sintaxis de las operaciones que meta-representan los módulos funcionales y de sistema son mostrados en la Figura 1.

```
sorts FModule Module .
subsort FModule < Module .
op fmod_is_sorts_._ _ _ _endfm : Qid ImportList SortSet SubsortDeclSet
                    OpDeclSet MembAxSet EquationSet -> FModule
                    [ctor gather (& & & & & & &)] .
op mod_is_sorts_._ _ _ _ _endm : Qid ImportList SortSet SubsortDeclSet
                    OpDeclSet MembAxSet EquationSet RuleSet -> Module
                    [ctor gather (& & & & & & & &)] .
```

Figura 1: Operadores que meta-representan los módulos

En la Figura 2 se muestran la signatura de las operaciones que extraen de la meta-representación de un módulo, la meta-representación de su nombre, de los módulos importados, y de la declaración de *sorts*, *subsorts*, operaciones, *memberships*, ecuaciones y reglas.

```
op getName : Module -> Qid .
op getImports : Module -> ImportList .
op getSorts : Module -> SortSet .
op getSubsorts : Module -> SubsortDeclSet .
op getOps : Module -> OpDeclSet .
op getMbs : Module -> MembAxSet .
op getEqs : Module -> EquationSet .
op getRls : Module -> RuleSet .
```

Figura 2: Operaciones que dan acceso a la metainformación de los módulos

3.3 Uso de la Reflexión en el Álgebra MOMENT

Una vez hemos visto las bases sobre las que se apoya la reflexión en *Maude*, se presenta cómo se puede aplicar al álgebra de MOMENT para dar más libertad al usuario de esta plataforma. Debemos tener en cuenta que estas especificaciones han sido realizadas para ilustrar el uso de la reflexión en MOMENT.

Como ya se ha comentado las limitaciones más fuertes que impone esta álgebra a la hora de manipular los modelos vienen dadas por las precondiciones que se establecen en cada operador. Con el uso de la reflexión podemos eliminar alguna de estas restricciones para dotar al usuario de mayor libertad, dejando a él la responsabilidad de trabajar con un sistema que contenga las inconsistencias que se pueden crear cuando no se satisfacen estas restricciones.

A continuación se indica cómo se puede aplicar la capacidad reflexiva de Maude para manipular la operación *safeDestroy* para el sort *Schema* del álgebra MOMENT. El módulo *Maude* que implementa esta operación tendría una forma parecida a la que se muestra en la Figura 3. La meta-representación del módulo de la Figura 3 sería la mostrada en la Figura 4.

```
(fmod SCHEMA is ...
    op schemaDestroyPC1___ : SchemaId LayerId ListLayer -> Bool .
    op schemaDestroyPC2___ : SchemaId LayerId ListLayer -> Bool .
    op safeDestroy_in__ : SchemaId LayerId ListLayer -> ListLayer .
    ...
    *** Ecuaciones que comprueban las precondiciones.
    eq schemaDestroyPC1 SI LI LL =
            not (existsInstanceOfSchema SI in (getLowerLayerOf LI LL)) .
    eq schemaDestroyPC2 SI LI LL = exists SI in (getLayer LI from LL) .
    *** Ecuación que realiza la eliminación "segura".
    ceq safeDestroy SI in LI LL =
            replace LI
            with (layer LI (delete SI
                                from layerGetListSchema(getLayer LI from LL)))
                                in LL
            *** Esta condición es la que comprueba las dos precondiciones.
                if schemaDestroyPC1 SI LI LL and schemaDestroyPC2 SI LI LL .

    ceq safeDestroy SI in LI LL = LL
            if not (schemaDestroyPC1 SI LI LL and schemaDestroyPC2 SI LI LL) .
    ...
endf)
```

Figura 3: Módulo que implementa la operación *safeDestroy* del *sort Schema*

```
fmod ‘SCHEMA is ...
    op ‘schemaDestroyPC1___ : ‘SchemaId ‘LayerId ‘ListLayer -> ‘Bool [none] .
    op ‘schemaDestroyPC2___ : ‘SchemaId ‘LayerId ‘ListLayer -> ‘Bool [none] .
    op ‘safeDestroy_in__ : ‘SchemaId ‘LayerId ‘ListLayer -> ‘ListLayer [none] .
    none
    eq ‘schemaDestroyPC1___[‘SI:SchemaId, ‘LI:LayerId, ‘LL:ListLayer] =
            ‘not ‘existsInstanceOfSchema_in_[‘SI:SchemaId,
                                ‘getLowerLayerOf__[‘LI:LayerId, ‘LL:ListLayer]] [none] .
    eq ‘schemaDestroyPC2___[‘SI:SchemaId, ‘LI:LayerId, ‘LL:ListLayer] =
                    ‘exists_in_[‘SI:SchemaId,
                        ‘getLayer_from:[‘LI:LayerId, ‘LL:ListLayer]] [none]  .
    ceq ‘safeDestroy_in_[‘SI:SchemaId ‘LI:LayerId ‘LL:ListLayer] =
       ‘replace_with_[‘LI:LayerId,
                ‘layer__[‘LI:LayerId,
                        ‘delete_from_in_[‘SI:SchemaId,
                                layerGetListSchema_[getLayer_from_[‘LI:LayerId,‘LL:ListLayer]],
                                ‘LL:ListLayer]]
                    if ‘schemaDestroyPC1___[‘SI:SchemaId, ‘LI:LayerId, ‘LL:ListLayer]
                     and ‘schemaDestroyPC2___[‘SI:SchemaId, ‘LI:LayerId, ‘LL:ListLayer] .
    ...
endf
```

Figura 4: Meta-representación del la operación *safeDestroy* del sort *Schema*

Se puede apreciar que la meta-representación de la operación safeDestroy introduce un mayor grado de complejidad, pero a cambio permite manipular la especificación con total libertad. Se puede, por ejemplo, eliminar las comprobaciones para dejar al usuario hacer pruebas, aún sabiendo que puede estar trabajando con un sistema inconsistente. Se puede incluso añadir nuevas ecuaciones que mantengan esta consistencia al eliminar las precondiciones, etc. Para reemplazar, por ejemplo, la primera precondición se precedería como sigue.

```
*** Definimos la operación para reemplazar ecuaciones en un módulo.
op reemplazar__ : FModule Equation -> FModule .
*** Esta será la nueva precondición.
op newPC1 : -> Ecuation .
var MID : Qid .              var IL : ImportList .          var SS : SortSet .
var SDS : SubsortDeclSet .   var ODS : OpDeclSer .          var MAS : MembAxSet .
var ES : EquationSet .               var E : Equation .

*** Se define la nueva precondición como true.
eq newPC1() =
   eq ‘schemaDestroyPC1___[‘SI:SchemaId, ‘LI:LayerId, ‘LL:ListLayer] =
                                                              ‘true.Bool [none] .
eq reemplazar(fmod ID is IL SS . SDS ODS MAS ES endfm, E)
    =  fmod ID is IL SS . SDS ODS MAS replace (ES, E) endfm .
```

Figura 5: Código para reemplazar ecuaciones en un módulo

En la Figura 5 se indica el código que define la ecuación para la nueva precondición, y la operación que reemplaza una ecuación en un módulo. Como se muestra, la nueva precondición siempre es verdadera, por lo que su efecto será el mismo que si la eliminaramos. Si suponemos que *mSchema* es el témino *Maude* que representa el meta-módulo, la reducción para reemplazar la precondición sería la que se muestra en la Figura 6. Por último, la Figura 7 muestra la nueva meta-representación del módulo *Schema* tras esta reducción.

```
reduce reemplazar(mSchema, newPC1()) .
```

Figura 6: Reducción que eliminaría la precondición

```
fmod 'SCHEMA is ...
    op 'schemaDestroyPC1___ : 'SchemaId 'LayerId 'ListLayer -> 'Bool [none] .
    op 'schemaDestroyPC2___ : 'SchemaId 'LayerId 'ListLayer -> 'Bool [none] .
    op 'safeDestroy_in__ : 'SchemaId 'LayerId 'ListLayer -> 'ListLayer [none] .
    none
    eq 'schemaDestroyPC1___['SI:SchemaId, 'LI:LayerId, 'LL:ListLayer]
                                                        = 'true.Bool [none] .
    eq 'schemaDestroyPC2___['SI:SchemaId, 'LI:LayerId, 'LL:ListLayer] =
                'exists_in_['SI:SchemaId,
                    'getLayer_from:['LI:LayerId, 'LL:ListLayer]] [none]  .
    ceq 'safeDestroy_in_['SI:SchemaId 'LI:LayerId 'LL:ListLayer] =
       'replace_with_['LI:LayerId,
              'layer__['LI:LayerId,
                    'delete_from_in_['SI:SchemaId,
                          layerGetListSchema_[getLayer_from_['LI:LayerId,'LL:ListLayer]],
                          'LL:ListLayer]]
                 if 'schemaDestroyPC1___['SI:SchemaId, 'LI:LayerId, 'LL:ListLayer]
                  and 'schemaDestroyPC2___['SI:SchemaId, 'LI:LayerId, 'LL:ListLayer] .
    ...
endf
```

Figura 7: Nueva meta-representación del la operación safeDestroy del sort Schema

4 Conclusiones y Trabajos Futuros

Cada vez más, los sistemas de información están constituidos por complicadas implementaciones donde tecnologías heterogéneas toman parte para solucionar aspectos específicos del sistema, las cuáles tienen que ser bien entendidas para ser manipuladas. El desarrollo de artefactos software involucra modelos, los cuáles pueden ser combinados o interconectados con otros con el fin de garantizar la interoperabilidad en un entorno distribuido, así como garantizar sus implementaciones.

La gestión de modelos es una disciplina emergente cuyo objetivo consiste en la integración de modelos de datos y la interoperabilidad entre artefactos software mediante operadores genéricos. El enfoque algebraico de la plataforma de gestión de modelos MOMENT, permite abordar el problema de una forma genérica utilizando sistemas de reescritura de términos para realizar dichas transformaciones de una forma sencilla puesto que la implementación consiste únicamente en definir un conjunto de reglas de reescritura.

En este trabajo se ha presentado parte de la teoría algebraica que utiliza MOMENT, expresada en CafeOBJ, para representar la estructura de la plataforma, así como los artefactos software que se pueden definir en cada uno de los niveles de abstracción de la plataforma MOMENT. Sobre esta teoría algebraica el uso de la reflexión de *Maude* aporta un nivel más de abstracción, permitiendo modificar las especificaciones. A partir de las especificaciones de la teoría MOMENT, podemos conseguir que evolucione manteniendo una uniformidad entre todas las capas, e incluso introduciendo nuevas.

Como trabajo inmediato se procederá a la traducción de la especificación algebraica de la teoría MOMENT a Maude, con el fin de que pueda ser manipulada mediante los mecanismos de reflexión que proporciona dicho entorno, así como aprovechar sus propiedades para llevar a cabo tareas de validación de modelos respecto a los metamodelos y gestionar su evolución. Los mecanismos de reflexión de Maude permiten gestionar la relación de instanciación entre los artefactos software que residen en los diferentes niveles de abstracción de la plataforma.

Actualmente, la teoría algebraica de MOMENT permite representar artefactos software en la plataforma pero no permite expresar relaciones entre ellos ni llevar a cabo transformaciones, tales como

combinación de modelos, composición, diferencia, etc. Dichas tareas, típicas en la disciplina de la gestión de modelos serán recogidas en una teoría que ampliará la existente mediante operadores genéricos que permitan expresar dichas manipulaciones y llevarlas a cabo sobre el sistema de reescritura *Maude*.

Como trabajo futuro también se realizará la integración de la herramienta de gestión de modelos en un entorno CASE de modelado que proporcione una buena interfaz visual para tratar con especificaciones algebraicas desde un punto de vista cercano al usuario. La herramienta también dispondrá de la funcionalidad de importar/exportar teorías basadas en lógica de reescritura a/de un formato común de intercambio de información entre diferentes sistemas de reescritura de términos. Esta funcionalidad permitirá utilizar diferentes sistemas de reescritura para llevar a cabo las manipulaciones de modelos, dependiendo de las propiedades que estos ofrezcan.

Referencias

[1] Artur Boronat, José Á. Carsí, Isidro Ramos, *Una plataforma para la Gestión Formal de Modelos*, Informe técnico. Ref. No: DSIC-II/4/04. Pag: 244. Julio 2004.

[2] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Carolyn Talcott (2003). ***"Maude 2.1.1 Manual. Versión 1.0"***. Web: *http://maude.csl.sri.com/*

[3] Boronat A. Carsí J. A., Ramos I., An Algebraic Baseline for Automatic Transformations in MDA. Workshop Software Evolution Through Transformations: Model-based vs. Implementation-level Solutions (SETra'04), 2nd International Conference on Graph Transformation (ICGT2004), ENTCS, Roma (Italia). Octubre 2004.

[4] Narciso Martí-Oliet and José Meseguer, *Rewriting Logic: Roadmap and Bibliography*, Theoretical Computer Science, Volume 285 , Issue 2 (August 2002). Pages: 121 – 154. ISSN: 0304-3975

[5] Object Management Group, *Model Driven Architecture (MDA)*, ormsc/01-07-01, July 2001. Available at http://www.omg.org/cgi-bin/doc?ormsc/01-07-01

[6] Diaconescu R., Futatsugi K., Ishisone M., Nakagawa A. T. y Sawada T. *An Overview of CafeObj*. In: C. Kirchner and H. Kirchner (eds), Proceedings of WRLA '98, Electronic Notes in Theoretical Computer Science, Vol. 13, 1998.

[7] OMG (2003), ***"MOF 1.4"***. Web: www.omg.org/mof

[8] OMG, Request for Proposal: MOF 2.0/QVT, OMG Document, ad/2002-04-10. Available at http://www.omg.org/cgi-bin/doc?ad/2002-04-10