

Un acercamiento a la representación e implementación de la diferencia entre modelos

Jose E. Rivera, Antonio Vallecillo

Dept. Lenguajes y Ciencias de la Computación

ETS Ingeniería Informática

Universidad de Málaga

29071 Málaga

{rivera,av}@lcc.uma.es

Resumen

En la Ingeniería Dirigida por Modelos (*Model Driven Engineering*, MDE), los modelos se convierten en los artefactos claves del desarrollo software. Estos modelos, al estar sometidos a un continuo proceso de evolución, hacen que ciertas operaciones como la diferencia estén adquiriendo cada vez más mayor importancia. En este artículo proponemos el uso de Maude como una notación formal para la descripción de modelos y metamodelos, e implementamos en él una definición de la diferencia entre modelos (y sus correspondientes operaciones) independiente del metamodelo al cual sean conformes.

1. Introducción

En el Desarrollo de Software Dirigido por Modelos (DSDM), los modelos pasan a ser los artefactos claves en todas las fases de desarrollo, desde la especificación del sistema y análisis, hasta el diseño y la implementación. Dentro de él se encuentra el Modelado Específico de Dominio (*Domain Specific Modeling*, DSM) como una manera de diseñar y desarrollar sistemas que implican el uso de Lenguajes Específicos de Dominio (*Domain Specific Language*, DSL) para la representación de varios aspectos de un sistema en términos de modelos. Con estos lenguajes se pretende conseguir una mayor abstracción que en los lenguajes de modelado de propósito general, y están más cercanos

al dominio del problema que al dominio de la implementación.

Hasta ahora, en DSDM la mayoría de los esfuerzos han sido invertidos en la definición de modelos, metamodelos y transformaciones entre ellos. Pero con su creciente adopción, está surgiendo la necesidad de disponibilidad de nuevas operaciones, tales como el subtipado de modelos [18], la inferencia de tipos, o la de creciente interés, diferencia entre modelos.

La diferencia entre modelos es una operación crucial en ciertos panoramas del DSDM. Entre ellos cabe destacar la gestión de versiones y el análisis de la evolución software, y las técnicas de comprobación en las que se comparan el resultado producido con el resultado esperado. Actualmente, las principales técnicas de visualización y representación de diferencias entre modelos son técnicas de coloreado [15] y *edit script* [1, 13]. El problema es que estas técnicas, o no obtienen como resultado un modelo (para poder así integrarlo en el proceso de DSDM), o no cumplen otras interesantes características tales como la composición [4]. Existen también otros trabajos que están basado simplemente en comparaciones textuales o de estructuras de datos [7, 8, 9, 10], o que sí están enfocados a modelos pero con la restricción de que éstos sean conformes a UML [1, 14, 19].

Lo que presentamos en este artículo es una definición de la diferencia entre modelos independiente del metamodelo al que sean con-

formas. Como resultado de esta diferencia, obtendremos un modelo conforme al que nosotros denominamos *Metamodelo de Diferencia*. Tanto la operación como el metamodelo, han sido (1) implementados en Maude aprovechando la notación formal propuesta en [17] para la representación de modelos y metamodelos, y (2) añadidos junto con sus operaciones relacionadas (Sección 5) a nuestro entorno desarrollado en Eclipse [16].

La estructura del documento es la siguiente. En primer lugar, la Sección 2 sirve como una breve introducción a Maude. A continuación, la Sección 3 describe cómo se pueden representar los modelos y los metamodelos en Maude. En la Sección 4 presentamos nuestra definición de la diferencia entre modelos y el *Metamodelo de Diferencia* al cual el modelo resultado será conforme. La Sección 5 introduce otras operaciones relacionadas con la diferencia y su implementación en Maude. Y para finalizar, la Sección 6 compara nuestro trabajos con aquellos relacionados y en la Sección 7 extraemos las conclusiones.

2. Lógica de reescritura y Maude

Maude [5, 6] es un lenguaje de alto nivel e intérprete y compilador de alto rendimiento de la familia de especificación algebraica de OBJ. Maude soporta lógica ecuacional de pertenencia y especificación y programación de lógica de reescritura de sistemas, integrando un estilo ecuacional de programación funcional con computación de lógica de reescritura. Debido a su máquina de reescritura, capaz de ejecutar más de tres millones de pasos de reescritura por segundo en un PC estándar, y debido a sus capacidades del metalenguaje, Maude resulta ser una excelente herramienta para crear entornos ejecutables para varias lógicas, modelos de computación, probadores de teoremas, o incluso lenguajes de programación. Además, Maude ha sido usado con éxito en aplicaciones de herramientas de Ingeniería del Software [12]. En esta sección, describiremos de manera informal aquellos elementos de Maude necesarios para entender el artículo; el lector que esté interesado puede acceder al manual [6] para

más detalles.

Full Maude es una extensión del lenguaje Maude que incluye entre otras propiedades una notación para la programación orientada a objetos. En él, los sistemas concurrentes orientados a objetos se especifican con módulos orientados a objetos donde se declaran clases y subclases. Una clase se declara con la sintaxis `class C | $a_1 : S_1, \dots, a_n : S_n$` , donde C es el nombre de la clase, a_i los identificadores de atributo, y S_i los tipos de los correspondientes atributos. Los objetos de una clase C son estructuras del estilo de registros de la forma `< $O : C | a_1 : v_1, \dots, a_n : v_n$ >`, donde O es el nombre del objeto, y v_i son los valores actuales de sus atributos. Los objetos pueden interactuar de varias maneras, incluyendo entre ellas el paso de mensaje. Los mensajes se declaran en Maude como cláusulas `msg`, en las que se definen su sintaxis y argumentos.

En un sistema concurrente orientado a objetos, el estado actual, el cual es llamado *configuración*, tiene la estructura de un multiconjunto compuesto por objetos y mensajes que evolucionan por reglas de reescritura concurrentes que describen el efecto de los eventos de comunicación de objetos y mensajes. El tipo predefinido `Configuration` representa una configuración de objetos y mensajes Maude, con `none` como una configuración vacía y el operador de sintaxis vacío `__` como la unión de configuraciones.

```
sort Configuration .
subsorts Object Message < Configuration .
op none : -> Configuration [ctor] .
op _ _ : Configuration Configuration
-> Configuration [ctor assoc comm id: none] .
```

En los módulos orientados a objetos, las reglas de reescritura son las que definen las transiciones entre configuraciones (para conocer detalladamente su comportamiento, referirse a [6]).

La herencia de clase está sustentada directamente por la estructura de tipos ordenadas de Maude. La declaración de una subclase `C` `<C'` indicando que C es una subclase de C' , es un caso particular de la declaración de subtipo `C <C'`, en la que todos los atributos mensajes, y regla de las superclases, así como los nuevos atributos definidos, mensajes y reglas de

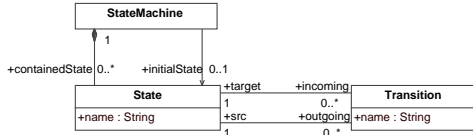


Figura 1: Metamodelo de Máquinas de Estado Simples.

la subclases caracterizan su estructura y comportamiento. Maude también soporta herencia múltiple [5].

3. Formalización de modelos y metamodelos usando Maude

Todavía no existe un acuerdo definitivo en la noción de modelo, por esa razón, nosotros adoptaremos la definición aportada por Jean Bézivin y Frederic Jouault en [2]: un modelo M es un triplete $M = (G, \omega, \mu)$ en el que G es un grafo multidirigido, ω es el *modelo referencia* de M (es decir, su metamodelo), y μ es una función que asocia los elementos (nodos y arcos) de G a los nodos del multigrafo que define ω (μ define la relación *conformeA* entre un modelo y su metamodelo).

Existen varias notaciones para representar modelos y metamodelos, desde textuales a gráficas. Una de interés particular para nosotros es KM3, un lenguaje textual especializado para la especificación de metamodelos, y cuya sintaxis abstracta está basado en Ecore y MOF 2.0. Así, KM3 tiene una terminología muy similar a la de Ecore y contiene las nociones de paquete, clase, atributo, referencia y tipo de dato. A continuación mostramos una posible especificación KM3 de un metamodelo para Máquinas de Estado Simples, el cual está también representado en la Figura 1.

```
package SimpleStateMachine {
class State {
  attribute name : String;
  reference stateMachine : StateMachine
    oppositeOf containedStates;
  reference incoming [*] : Transition
    oppositeOf target;
  reference outgoing [*] : Transition
    oppositeOf src;
}
```

```
}
class StateMachine {
  reference initialState [0-1] : State;
  reference containedStates [*] container : State
    oppositeOf stateMachine;
}
class Transition {
  attribute name : String;
  reference target [1] : State
    oppositeOf incoming;
  reference src [1] : State
    oppositeOf outgoing;
}
}
```

Existen muchos beneficios asociados al uso de KM3, como por ejemplo: es sencillo y fácil de aprender y entender; permite definiciones y modificaciones de metamodelos fáciles y precisas; se pueden transformar MOF, Ecore y otros lenguajes de metamodelos desde/a las descripciones KM3; está bien soportado por herramientas y está integrado con el entorno AMMA (*ATLAS Model Management Architecture*, <http://www.sciences.univ-nantes.fr/lina/atl/>). Además, los metamodelos definidos en KM3 se pueden incluir en repositorios de modelos (zoos) para la práctica de mega-modelado [3].

Sin embargo, KM3 no es la única notación para la descripción de modelos y metamodelos. En [17] presentamos una propuesta basada en Maude, que además de ser lo suficientemente expresiva, ofrece buen soporte para razonar acerca de los modelos. Concretamente, enseñamos cómo algunas operaciones básicas sobre modelos, como el subtipado de modelos o la inferencia de tipos, se pueden especificar e implementar fácilmente en Maude, y poner a disposición del usuario en entornos de desarrollo como Eclipse. En esta sección, presentamos sólo un pequeño resumen de dicha propuesta.

3.1. Representación de modelos y metamodelos en Maude

En Maude, los modelos se representan con configuraciones de objetos. Un nodo por lo tanto, estará representado con un objeto Maude. Tanto los atributos que puede tener un nodo, como los arcos entre nodos, se representarán con los atributos de los objetos Maude. Concretamente, los arcos entre nodos repre-

sentarán cada uno una referencia al nodo destino del arco.

Debido a la manera en que representamos los modelos, existen dos posibles formas de representar los metamodelos. En primer lugar, podemos representar un metamodelo como un módulo orientado a objetos Maude que contenga la especificación de un conjunto de clases. Estas clases se corresponderán con aquellas a las cuales pertenezcan los objetos que representan los correspondientes nodos de los modelos. De esta forma, los modelos son conformes a sus metamodelos por construcción.

En segundo lugar, ya que los metamodelos son también modelos, se pueden representar como configuraciones de objetos. Las clases de estos objetos serán aquellas que especifiquen los meta-metamodelos, como por ejemplo, las clases definidas en el metamodelo de KM3.

Para ilustrar la primera opción de representación de los metamodelos, el siguiente módulo de Maude describe el metamodelo de Máquinas de Estado Simples.

```
(omod SimpleStateMachines is
  protecting STRING .
  class State |
    name : String,
    stateMachine : Oid,
    incoming : Set{Oid},
    outgoing : Set{Oid} .
  class StateMachine |
    containedStates : Set{Oid},
    initialState : Maybe{Oid} .
  class Transition |
    name : String,
    target : Oid,
    src : Oid .
endom)
```

Las clases de KM3 se corresponden con las clases Maude. Los atributos se representan como atributos Maude. Las referencias como atributos Maude también, pero en términos de identificadores de objeto Maude (*Object identifier*, `Oid`). Dependiendo de la multiplicidad, podemos usar: un simple identificador (si la multiplicidad es 1); un `Maybe{Oid}` que representa un identificador o un valor nulo (`null`), para especificar la multiplicidad [0-1]; un conjunto de identificadores (`Set{Oid}`) para la multiplicidad [*]; o una lista de identificadores (`List{Oid}`) en el caso de que las referencias estén ordenadas. Nótese que en esta

```
< 'SM : StateMachine | initialState : 'ST1,
  containedStates : ('ST1, 'ST2) >
< 'ST1 : State | name : "St1", stateMachine : 'SM,
  outgoing : 'TR, incoming : empty >
< 'ST2 : State | name : "St2", stateMachine : 'SM,
  incoming : 'TR, outgoing : empty >
< 'TR : Transition | name : "Tr", src : 'ST1,
  target : 'ST2 >
```

Figura 2: Modelo de una Máquina de estados.

forma de representación no se tienen en cuenta algunos aspectos de KM3, como las referencias opuestas (`oppositeOf`). Ésta y otras nociones de KM3, se recogen en la forma alternativa de representación de los metamodelos (como configuración de objetos).

Las instancias de tales clases representarán modelos conformes al metamodelo descrito. Por ejemplo, la configuración de objetos Maude mostrada en la Figura 2, representa un modelo de una máquina de estados con dos estados, llamados `St1` y `St2`, y una transición (`Tr`) entre ellos. `St1` es el estado inicial de la máquina de estados.

La validez de los objetos en una configuración es comprobada por el sistema de tipos de Maude. Otros aspectos del metamodelo, como la validez de los tipos de los objetos referenciados y las referencias opuestas (`opposite`), se expresan en Maude en términos de axiomas de pertenencia que definen las reglas bien formadas que cualquier modelo válido debe cumplir [17].

Nuestra segunda manera de representar los metamodelos es considerándolos modelos, y por lo tanto, también se pueden representar como configuración de objetos. Las clases de tales objetos serán aquellas especificadas en el metametamodelo—por ejemplo, las clases definidas en el metamodelo KM3.

Estas dos formas de representación no son completamente equivalentes. De hecho, la segunda contiene toda la información acerca del metamodelo, mientras que la primera describe sólo la información necesarias sobre los modelos en sí para poder instanciarlos ([17]).

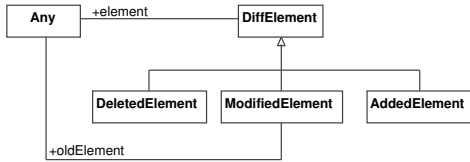


Figura 3: Metamodelo de Diferencia.

4. Diferencia entre modelos

Una vez descrito como los modelos y los metamodelos se pueden representar en Maude, en esta sección vamos a presentar una posible definición de diferencia entre modelos, y su implementación en Maude.

4.1. Metamodelo de Diferencia

Como comentábamos en la introducción, estamos interesados en que la diferencia entre modelos tenga como resultado otro modelo, para poder integrarlo así en el proceso de DSDM. De esa forma, lo primero que hay que hacer es definir un metamodelo en el que se describa los elementos que una diferencia puede contener, así como las relaciones entre ellos. Además, nuestro *Metamodelo de Diferencia* tiene que ser lo suficientemente general para que sea independiente del metamodelo con respecto al cual sean conformes los modelos a comparar.

Siguiendo estas pautas, hemos desarrollado el *Metamodelo de Diferencia* representado en la figura 3. Un modelo diferencia contendrá los cambios realizados de un modelo sustraendo a un modelo minuendo. En general, podemos distinguir tres tipos básicos de cambio: adición de un elemento, supresión de un elemento, o modificación de un elemento. Por lo tanto, todo elemento del modelo diferencia (`DiffElement`) tendrá como metaclasses `DeletedElement`, `ModifiedElement` o `AddedElement`, según se haya borrado, modificado, o añadido un elemento, respectivamente. Los elementos que no sufran cambio alguno, no quedarán reflejados en el modelo diferencia.

Todo elemento de la diferencia `DiffElement` tendrá una referencia (`element`) al elemen-

to en cuestión que ha sufrido el cambio. En el caso de la modificación de un elemento (`ModifiedElement`), se tendrá una referencia tanto al elemento del modelo minuendo (posterior a la modificación, `element`), como al del modelo sustraendo (anterior a la modificación, `oldElement`). Los elementos que han sufrido el cambio, también se almacenan en el modelo diferencia. Éstos pueden tener cualquier metaclasses, pues la diferencia se aplica a un par de modelos conforme a un metamodelo cualquiera. Así, hemos introducido la metaclasses `Any`, a la que cualquier elemento objeto pertenece.

La representación del *Metamodelo de Diferencia* en Maude quedaría de la siguiente forma:

```

(omod ModelDiff is
  class Any .
  class DiffElement | element : Oid .
  class AddedElement .
  class DeletedElement .
  class ModifiedElement | oldElement : Oid .

  subclasses AddedElement DeletedElement
              ModifiedElement < DiffElement .
endom)

```

La clase `Any` se puede implementar en Maude de forma natural haciendo uso del identificador de clase ya definido `Cid` (*Class identifier*, [6]).

4.2. La operación de diferencia en Maude

Dado un modelo minuendo M_m y un modelo sustraendo M_s , conformes a un metamodelo cualquiera, el resultado de aplicar la operación de diferencia entre modelos a ellos dará como resultado otro modelo M_d conforme al *Metamodelo de Diferencia*, de forma que $\text{modelDiff}(M_m, M_s) = M_d$.

La declaración de esta operación en Maude quedaría de la siguiente forma:

```

subsort DiffModel < Configuration .
op modelDiff : Configuration Configuration
-> DiffModel .

```

A la hora realizar la diferencia entre dos modelos, son cuatro los casos principales que podemos encontrar: (1) que un elemento esté en ambos modelos y no haya sido modificado, (2) que esté en ambos y sí haya sido modificado,

(3) que sólo esté en el modelo minuendo, o (4) que sólo esté en el modelo sustraendo. Para cada uno de estos casos, hemos implementado la correspondiente ecuación Maude que lo contempla. En todas ellas usaremos las variables `O1` y `O2` de tipo `Obj`, `C`, `C1` y `C2` de tipo `Cid`, `ATTS`, `ATTS1` y `ATTS2` de tipo `AttributeSet`, y `CONF1` y `CONF2` de tipo `Configuration`.

En el primer caso, tenemos que comprobar que dos objetos (uno del modelo minuendo y otro del modelo sustraendo) emparejen, es decir, que representen el mismo elemento, y que los dos pertenezcan a la misma clase y posean los mismos valores en sus atributos (recordamos que en Maude tanto las referencias como los atributos se expresan mediante atributos Maude). Si esto ocurre, es porque el elemento no ha sido modificado, y en la diferencia no quedará constancia de él.

```
ceq modelDiff(< O1 : C | ATTS > CONF1,
              < O2 : C | ATTS > CONF2)
  = modelDiff(CONF1,CONF2)
  if match(< O1 : C | ATTS >, < O2 : C | ATTS >)
```

Nótese que para indicar que los dos objetos pertenecen a la misma clase y tienen los mismos valores en los atributos, usamos, respectivamente, las mismas variables `C` y `ATTS` en ambos.

En Maude, como los objetos tienen identificadores, para saber que dos objetos representan el mismo elemento basta con comprobar que tienen el mismo identificador. Esto no ocurre en otras formas de representación de modelos. Por lo tanto, y debido a las transformaciones definidas en ATL desde otras formas de representación a Maude [17], hemos creado la función `match`, para que se pueda modificar según convenga.

Sólo hay que sustituir la ecuación actual que compara los identificadores de dos objetos por aquella que deseamos, como por ejemplo, la función de emparejamiento estructural definido en [11].

```
op match : Object Object -> Bool .
eq match(< O1 : C1 | ATTS1 >, < O2 : C2 | ATTS2 >)
  = (O1 == O2) .
```

Estas funciones suelen ser bastante sencillas de especificar en Maude, gracias al emparejamiento y reordenación que nos ofrece sobre

las configuraciones. Por ejemplo, si quisiéramos que dos objetos emparejaran únicamente cuando pertenezcan a la misma clase y tengan el mismo nombre (un atributo nombre, suponiendo que lo tienen), dado las variables `C` de tipo `Cid`, `N` de tipo `String`, y `OBJ1` y `OBJ2` de tipo `Object`, las ecuaciones a añadir quedarían definidas de la siguiente forma:

```
eq match(< O1 : C | nombre : N, ATTS1 >,
         < O2 : C | nombre : N, ATTS2 >)
  = true .
eq match(OBJ1, OBJ2) = false [owise] .
```

El segundo caso, es aquel en el que dos objetos se refieren a un mismo elemento, pero éste ha sido modificado. Es decir, ambos objetos emparejan, pero tienen distintos valores en sus atributos (atributos Maude) o pertenecen a distinta clase (en Maude se permite la reclasificación de un objeto de manera dinámica). En este caso, crearemos un objeto de tipo `ModifiedElement` con referencias al objeto del modelo sustraendo (previo a la modificación, `oldElement`) y al objeto del modelo minuendo (posterior a la modificación, `element`). Ambos objetos se añadirán también al modelo diferencia, pero sólo con los atributos necesarios. Estos atributos son aquellos que están definidos en ambos objetos pero con distinto valor (de esta forma almacenaremos el valor anterior y el posterior a la modificación), o aquellos que están definidos en un objeto pero en el otro no (en el caso de los atributos del objeto del modelo sustraendo serán los atributos borrados, mientras que en los del modelo minuendo serán los añadidos). A ambos objetos añadidos se les modificará el identificador (con las funciones `newId` y `oldId`) para diferenciarlos entre sí, ya que en Maude todos los objetos de una misma configuración deben tener identificadores distintos. Las modificaciones sobre los identificadores se harán del tal forma que podamos deshacerlas para obtener los correspondientes identificadores iniciales (con la función `originalId`, usada en la siguiente sección).

```
ceq modelDiff(< O1 : C1 | ATTS1 > CONF1,
              < O2 : C2 | ATTS2 > CONF2)
  = < newModId(O1) : ModifiedElement |
    element : newId(O1), oldElement : oldId(O2) >
  < newId(O1) : C1 | attsDiff(ATTS1,ATTS2) >
  < oldId(O2) : C2 | attsDiff(ATTS2,ATTS1) >
```

```

modelDiff(CONF1,CONF2)
if match(< O1 : C1 | ATTS1 >,< O2 : C2 | ATTS2 >)
/\ (not (ATTS1 == ATTS2) or not (C1 == C2)) .

```

Los casos tercero y cuarto, son aquellos en los que un objeto de un modelo no empareja con ningún otro del modelo restante. Si el objeto se encuentra sólo en el modelo minuendo, es que el elemento ha sido añadido, mientras que si el objeto se encuentra sólo en el modelo sustraendo, es que el elemento ha sido eliminado. Así, crearemos un objeto `AddedElement` o `DeletedElement`, respectivamente, con una referencia al objeto en cuestión, el cual también añadiremos al modelo diferencia (con el identificador modificado, como en el caso anterior).

```

eq modelDiff( < O1 : C1 | ATTS1 > CONF1, CONF2)
= < newAddId(O1) : AddedElement |
  element : newId(O1) >
  < newId(O1) : C1 | ATTS1 >
  modelDiff(CONF1,CONF2) [owise] .

eq modelDiff( CONF1, < O2 : C2 | ATTS2 > CONF2)
= < newDelId(O2) : DeletedElement |
  element : oldId(O2) >
  < oldId(O2) : C2 | ATTS2 >
  modelDiff(CONF1,CONF2) [owise] .

```

El atributo `owise`, es el que precisamente impone que el objeto de un modelo no empareja con ninguno del otro, pues si no se ejecutaría la primera o la segunda ecuación (según corresponda).

Existe realmente un quinto caso, el caso base, en el que los dos modelos sustraendo y minuendo son vacíos. El resultado entonces, como es de esperar, también será el modelo diferencia vacío.

```

eq modelDiff( none, none) = none .

```

Para aclarar un poco más el funcionamiento de la operación, vamos a introducir un ejemplo. Dada la máquina de estados representada en la Figura 2, supongamos que le añadimos una transición en el sentido opuesto a la ya existente (una nueva transición `Tr2` del estado `St2` hacia `St1`). Como resultado, obtendremos el modelo representado en la Figura 4. Nótese que en este caso, los estados `St1` y `St2` también sufren una modificación por tener una referencia a las transiciones de entrada y salida.

```

< 'SM : StateMachine | initialState : 'ST1 ,
  containedState : ('ST1, 'ST2) >
< 'ST1 : State | name : "St1", stateMachine : 'SM,
  outgoing : ('TR, 'TR2), incoming : empty >
< 'ST2 : State | name : "St2", stateMachine : 'SM,
  incoming : ('TR, 'TR2), outgoing : empty >
< 'TR : Transition | name : "Tr", src : 'ST1 ,
  target : 'ST2 >
< 'TR2 : Transition | name : "Tr2", src : 'ST2 ,
  target : 'ST1 >

```

Figura 4: Modelo de la Máquina de estados modificado.

```

< 'ST1@MOD : ModifiedElement | element : 'ST1@NEW,
  oldElement : 'ST1@OLD >
< 'ST1@NEW : State | outgoing : ('TR, 'TR2) >
< 'ST1@OLD : State | outgoing : 'TR >
< 'ST2@MOD : ModifiedElement | element : 'ST2@NEW,
  oldElement : 'ST2@OLD >
< 'ST2@NEW : State | incoming : ('TR, 'TR2) >
< 'ST2@OLD : State | incoming : 'TR >
< 'TR2@ADDED : AddedElement | element : 'TR2@NEW >
< 'TR2@NEW : Transition | target : 'ST2,
  name : "Tr2", src : 'ST1 >

```

Figura 5: Ejemplo de Modelo diferencia en Maude.

Si tomamos este último modelo como minuendo, y el primero como sustraendo, el resultado de aplicarles la operación diferencia sería el modelo conforme al *Metamodelo de Diferencia* que se muestra en la Figura 5. Como podemos observar, en él queda reflejado tanto la adición de la transición `Tr2`, como la modificación de las correspondientes referencias de los estados `St1` y `St2`.

5. Operaciones complementarias

Una vez definida la diferencia entre modelos, son varias las nuevas operaciones que se pueden definir en torno a ella. Así, entre otras, hemos definido las operaciones `do` y `undo`, que nos permiten obtener el modelo minuendo a partir del sustraendo y viceversa, respectivamente.

5.1. Operación `do`

Dado un modelo M_s conforme a un metamodelo cualquiera MM , y un modelo M_a conforme al *Metamodelo de Diferencia*, el resultado de aplicar la operación `do` sobre ellos, será un

modelo M_m conforme al metamodelo MM , de forma que $\text{do}(M_s, M_d) = M_m$, y cumpliéndose que $\text{modelDiff}(M_m, M_s) = M_d$. Realmente, M_m será conforme al mismo metamodelo que M_s siempre que así fuera a la hora de realizar la diferencia, lo que es de esperar.

La operación **do** aplica a un modelo los cambios especificados en el modelo diferencia. En Maude, esta operación se puede expresar mediante tres ecuaciones, cada una de ellas correspondientes a la adición, supresión, y modificación de elementos. A continuación, mostramos la correspondiente especificación Maude.

```

vars MODEL CONF : Configuration .
vars O O2 OLDO NEWO : Oid .
vars C NEWC OLDC : Cid .
vars ATTS OLDATTS NEWATTS : AttributeSet .

op do : Configuration DiffModel
  -> Configuration .

eq do(MODEL, < O : AddedElement | element : NEWO >
  < NEWO : NEWC | NEWATTS > CONF)
= < originalId(NEWO) : NEWC | NEWATTS >
  do(MODEL, CONF) .

ceq do(< O : C | ATTS > MODEL,
  < O2 : DeletedElement | element : OLDO >
  < OLDO : OLDC | OLDATTS > CONF)
= do(MODEL, CONF)
  if match(< O : C | ATTS >,
    < originalId(OLDO) : OLDC | OLDATTS >) .

ceq do(< O : C | ATTS > MODEL,
  < O2 : ModifiedElement | element : NEWO,
  oldElement : OLDO >
  < NEWO : NEWC | NEWATTS >
  < OLDO : OLDC | OLDATTS > CONF)
= < originalId(NEWO) : NEWC |
  (excludingAll(ATTS, OLDATTS), NEWATTS) >
  do(MODEL, CONF)
  if match(< O : C | ATTS >,
    < originalId(OLDO) : OLDC | OLDATTS >) .

eq do(MODEL, none) = MODEL .

```

La función **originalId** recupera el identificador original del objeto que sufrió el cambio, es decir, deshace los cambios realizados en su identificador por la función **newId** u **oldId** aplicadas en la diferencia entre modelos. La función **excludingAll**, usada en la ecuación correspondiente a la modificación de elementos, elimina de un conjunto de atributos Maude **ATTS** todos aquellos que tengan el mismo nombre que alguno del conjunto **OLDATTS**. Ya que en **OLDATTS** se encuentran todos los atributos que tienen que ser eliminados, o que fueron

modificados (pero con el valor antiguo), lo que estamos haciendo precisamente es eliminarlos al conjunto **ATTS** todos ellos, para luego añadirles el conjunto **NEWATTS**, que contiene los atributos que tienen que ser añadidos, y los atributos que fueron modificados, pero con los nuevos valores.

Nótese que la función **do** está especificada de tal forma, que se puede aplicar sobre cualquier modelo independientemente que éste fuera o no el modelo sustraendo original de una diferencia.

5.2. Operación (undo)

Dado un modelo M_m conforme a un metamodelo cualquiera MM , y un modelo M_d conforme al *Metamodelo de Diferencia*, el resultado de aplicar la operación **undo** a ellos, será un modelo M_s conforme al metamodelo MM , de forma que $\text{undo}(M_m, M_d) = M_s$, y cumpliéndose que $\text{modelDiff}(M_m, M_s) = M_d$. Al igual que en la operación **do**, M_s será conforme al mismo metamodelo que M_m siempre que así fuera a la hora de realizar la diferencia, lo que es de esperar.

La operación **undo** es la función inversa de **do**, es decir, realiza las operaciones contrarias. Por lo tanto, sus ecuaciones son exactamente iguales a las ecuaciones de **do**, sólo que cambiando que los elementos que se añaden son los elementos **DeletedElement**, que los elementos que se eliminan son los **AddedElement**, y que a los atributos de los elementos modificados se les elimina los especificados en el objeto representante del modelo minuendo (**NEWATTS**), y se les añade los del sustraendo (**OLDATTS**), en vez de al contrario.

5.3. Otras operaciones

Además de las operaciones mencionadas **do** y **undo**, han sido definidas otras operaciones como la composición de modelos diferencia (para cuestiones de optimización en sucesivas modificaciones sobre un mismo elemento) y la inversa de un modelo diferencia (dado un modelo diferencia M_d , y otros dos modelos M_m y M_s tal que $\text{modelDiff}(M_m, M_s) =$

M_d , hemos denominado modelo diferencia inverso de M_d , a aquel modelo M'_d que cumple $\text{modelDiff}(M_s, M_m) = M'_d$.

6. Trabajos relacionados

Son varios los trabajos existentes relacionados con la diferencia entre modelos. En primer lugar, tenemos aquellos que están diseñados para la diferencia entre modelos conforme a un metamodelo concreto, como UML [1, 14, 19]. En segundo lugar, existen técnicas como la de coloreado [15] y *edit scripts* [1, 13] para la representación de la diferencia, pero cuyo resultado no es ajustable a un metamodelo, y por lo tanto no puede ser procesado por plataformas estándar de modelado [4].

Otros trabajos, como [4] y [11], están más estrechamente relacionados con el nuestro. En [4], se propone otra representación de la diferencia entre modelos independiente del metamodelo al que sean conformes, pero en este caso, el *Metamodelo Diferencia* es una ampliación del metamodelo al cual sean conformes los metamodelos operandos. Esta solución al ser agnóstica del método de cálculo, no introduce ninguna operación de diferencia, y además necesita la ejecución de numerosas transformaciones (definidas en ATL) tanto para la creación del Metamodelo Diferencia específico para el caso, como en la aplicación a un modelo cualquiera del modelo diferencia (operación *do*), perdiendo la simpleza (también traducida en tiempo de cómputo) que se consigue en nuestra solución. En [11], sí presentan un algoritmo de diferencia entre modelos, también independiente del metamodelo al que sean conformes, pero cuyo resultado no es compacto (contiene información adicional a la necesaria para representar la diferencia) y está más orientado a usarse para la representación de las diferencias.

7. Conclusión

En este artículo hemos presentado una posible definición de la diferencia entre modelos independiente del metamodelo al que sean

conformes. El resultado de aplicar esta operación, será un modelo conforme al *Metamodelo de Diferencia* introducido, siguiendo los principios de DSDM. Tanto el metamodelo, como la operación de diferencia, y otras operaciones complementarias, como aquellas para obtener el modelo minuyendo a partir del sustraendo y viceversa, han sido implementadas en Maude e integradas en nuestro entorno desarrollado en Eclipse [17]. Estas operaciones pueden ser aplicadas a modelos no especificados inicialmente en Maude, gracias a las transformaciones que hemos definido con ATL desde otras formas de representación tales como KM3 [16]. Así, y para que Maude quede totalmente transparente al usuario, actualmente estamos trabajando en las transformaciones en el sentido inverso.

Agradecimientos Este trabajo ha sido financiado por el proyecto TIN2005-09405-C02-01. También agradecemos a los revisores del trabajo sus cuidados comentarios, que nos han ayudado a mejorarlo.

Referencias

- [1] M. Alanen and I. Porres. Difference and union of models. Technical report, Turku Centre for Computer Science, Apr. 2003.
- [2] J. Bézivin and F. Jouault. KM3: a DSL for metamodel specification. In *Procs. of the 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006)*, volume 4037 of *LNCS*, pages 171–185, Bologna, Italy, Apr. 2006. Springer-Verlag.
- [3] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the large and modeling in the small. In U. Aßmann, M. Aksit, and A. Rensink, editors, *Model Driven Architecture, European MDA Workshops: Foundations and Applications (MDA-FA 2003/2004)*, volume 3599 of *LNCS*, pages 33–46. Springer-Verlag, 2005.
- [4] A. Cicchetti, D. D. Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. In

- B. Meyer and J. Bézivin, editors, *Proc. of TOOLS Europe 2007*, Zurich, Switzerland, June 2007.
- [5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Comput. Sci.*, 285:187–243, Aug. 2002.
- [6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Tallcott. *Maude 2.0 Manual*, June 2003. <http://maude.cs.uiuc.edu>.
- [7] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *Software Engineering, IEEE Transactions on*, 27(1):1–12, 2001.
- [8] S. G. Eick, J. L. Steffen, and J. Eric E. Sumner. Seesoft – a tool for visualizing line oriented software statistics. In *Readings in information visualization: using vision to think*, pages 419–430, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
- [9] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 23, Washington, DC, USA, 2003. IEEE Computer Society.
- [10] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.
- [11] Y. Lin, J. Gray, and F. Jouault. Dsmdiff: A differentiation tool for domain-specific models. Technical report, University of Alabama at Birmingham and University of Nantes, Dec. 2006.
- [12] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Comput. Sci.*, 285(2):121–154, 2002.
- [13] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.
- [14] D. Ohst, M. Welle, and U. Kelter. Difference tools for analysis and design documents. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.
- [15] D. Ohst, M. Welle, and U. Kelter. Differences between versions of uml diagrams. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering conference*, pages 227–236, New York, NY, USA, 2003. ACM Press.
- [16] J. E. Rivera, F. Durán, A. Vallecillo, and J. R. Romero. Maudeling: Herramienta de gestión de modelos usando maude. In *JISBD' 2007: Actas de XII Jornadas de Ingeniería del Software y Bases de Datos*, Sept. 2007.
- [17] J. R. Romero, J. E. Rivera, F. Durán, and A. Vallecillo. Formal and tool support for model driven engineering with maude. In B. Meyer and J. Bézivin, editors, *Proc. of TOOLS Europe 2007*, Zurich, Switzerland, June 2007.
- [18] J. Steel and J.-M. Jézéquel. Model typing for improving reuse in model-driven engineering. In L. Briand and C. Williams, editors, *Procs. of MoDELS 2005*, volume 3713 of *LNCS*, pages 84–96. Springer-Verlag, July 2005.
- [19] Z. Xing and E. Stroulia. Umlidiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, pages 54–65, New York, NY, USA, 2005. ACM Press.