ESTABLECIMIENTO Y VERIFICACIÓN DE LA CONSISTENCIA EN DOME: UN CASO DE ESTUDIO

CONSISTENCY ESTABLISHMENT AND VERIFICATION IN DOME: A CASE STUDY

DANIEL CABARCAS
Grupo de Investigación UN-INFO. Escuela de Sistemas. Facultad de Minas.Universidad Nacional de Colombia, sede Medellín. dcabarc@unalmed.edu.co

FERNANDO ARANGO
Grupo de Investigación UN-INFO. Escuela de Sistemas. Facultad de Minas.Universidad Nacional de Colombia, sede Medellín. . farango@unalmed.edu.co

CARLOS M. ZAPATA
Grupo de Investigación UN-INFO. Escuela de Sistemas. Facultad de Minas. Universidad Nacional de Colombia, sede Medellín. cmzapata@unalmed.edu.co

Recibido para revisar 11 de Febrero de 2005, aceptado 29 de Agosto de 2005, versión final 21 de Septiembre de 2005

RESUMEN: Las herramientas metaCASE ofrecen una funcionalidad similar a la de las herramientas CASE convencionales para notaciones gráficas arbitrarias, una vez estas notaciones le sean especificadas adecuadamente. La principal dificultad de dichas herramientas es la especificación de las diferentes reglas de consistencia, que deben tenerse en cuenta cuando se usa una notación. En este artículo se presenta la especificación de dos reglas de consistencia del diagrama de clases de UML en el metaCASE DOME, codificadas en el lenguaje de programación Alter. Adicionalmente, se hace un análisis comparativo entre las especificaciones de los aspectos estructurales y de las reglas de consistencia en DOME y en la especificación de UML provista por el OMG.

PALABRAS CLAVE: UML, Metamodelamiento, DOME, Consistencia.

ABSTRACT: MetaCASE tools offer a similar functionality than conventional CASE tools for arbitrary graphical notations, once these notations had been suitably specified. The main obstacle affronted when using these tools, is the specification of the consistency rules that must be taken into account when using the graphical notation. In this paper, we present the specifications of two consistency rules of the UML class diagram on the metaCASE tool DOME, encoded on the programming language *Alter*. In addition, we analyze not only the specification of the structural features, but also the specification of the consistency rules of the UML class diagram in DOME, comparing with the UML specification provided by OMG.

KEYWORDS: UML, Metamodeling, DOME, Consistency.

1. INTRODUCCIÓN

Diseñadores de todo tipo de sistemas se están apoyando cada vez más en modelos gráficos, no sólo para obtener una visión general del problema, sino para entender mejor las implicaciones de sus decisiones y, más aún, para aplicar transformaciones que reduzcan la brecha entre el diseño y la implementación, de manera que cada vez es más necesario comprobar la integridad de un modelo gráfico, para asegurar que el análisis posterior o un potencial paso de refinamiento entreguen un resultado confiable.

Para seguirle el paso a la gran cantidad de notaciones gráficas que aparecen día a día para apoyar el diseño, en los últimos años han surgido varias metaherramientas que permiten rápidamente generar herramientas para el modelamiento gráfico. Algunas de ellas son: AToM3 (De Lara y Vangheluwe 2002), MetaEdit+ (MetaEdit+ 2004), DOME (DOME 2004), GME (MetaGME 2004), DiaGen (DiaGen 2004), CodeSign (CodeSign 2004) y Moses (Esser y Janneck 2001). Este tipo de herramientas aparecen tratando de cumplir el papel que durante muchos años han cumplido generadores automáticos de *parsers* o de analizadores léxicos, en la generación automática de lenguajes de programación. En este último campo existe una notación unificadora representada por la Forma *Backus Naur*, una estructura jerárquica tradicionalmente usada para expresar las gramáticas textuales en un lenguaje de programación específico. En el metamodelamiento, empero, aún no existe una notación unificadora para la definición de una nomenclatura gráfica.

Entre estas metaherramientas aparece DOME (*Domain Modeling Environment*), una herramienta diseñada para especificar notaciones gráficas y generar los editores correspondientes. Se destaca porque sus especificaciones son orientadas a objetos y pueden ser interpretadas "on-the-fly" (se realizan los cambios directamente en el metamodelo y se pueden usar inmediatamente en la zona de edición de instancias). Además, DOME cuenta con muchos conceptos preconstruidos que aceleran el desarrollo de notaciones gráficas y es soportado por un poderoso "backend" para la generación de código y/o documentación (Engstrom y Krueger 2000).

En este artículo se estudian las capacidades y facilidades que ofrece DOME para verificar consistencia del diagrama de clases UML, mediante un caso de estudio. En la sección 2 se introduce DOME como herramienta de metamodelamiento y en la sección 3 se analiza una especificación en DOME que genera un editor de UML. En la sección 4 se discuten brevemente los mecanismos que ofrece DOME para especificar y verificar consistencia, con el fin de mostrar en la sección 5 la implementación de dos reglas de consistencia del metamodelo de UML sobre la especificación introducida en la sección 3.

2. DOMAIN MODELING ENVIRONMENT, DOME.

La versión 5.3 de DOME permite especificar una notación gráfica como un modelo llamado *DOME Tool Specification*, DTS (DOME, 2004). Para efectos prácticos una DTS es creada, editada e interpretada utilizando ProtoDOME, que es una herramienta visual. En una DTS se pueden distinguir dos componentes: una especificación de alto nivel, basada en nodos, atributos, conectores, entre otros elementos y un complemento a más bajo nivel, escrito en un lenguaje funcional basado en *Scheme*, un lenguaje de programación similar al *Lisp*, para implementar comportamientos especiales.

La especificación de los aspectos estructurales de una notación gráfica, parte de uno o varios gráficos (denotados como *Graph* en el entorno de DOME) que describen cada uno de los diagramas contemplados en la notación. Un *Graph* puede contener especificaciones de los siguientes tipos (véase la Figura 1 para apreciar la descripción gráfica de estos tipos en DOME): *Node*,

Este artículo se realizó en el marco del proyecto de Investigación "Extensiones en herramientas CASE con énfasis en formalismos y reutilización" financiado por COLCIENCIAS, la Universidad Nacional de Colombia y la Universidad EAFIT.

Connector, *List Element*, *Connection Constraint*, *Accessory part/whole*, *List Element part/whole*, *Node part/whole*, *Generic Specification*, *Basic Class*, *Property*, *Method* y *Relationship*. En el resto del artículo, por simplicidad, una especificación de tipo *Node* se denominará "un *Node*", y así sucesivamente.

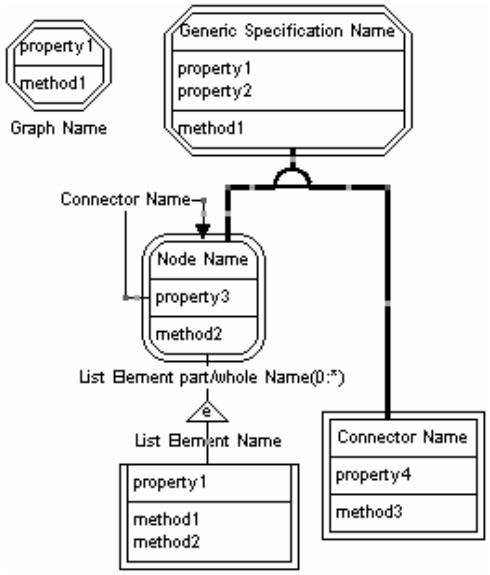

Figura 1. Los principales elementos de una DTS.
Figure 1. Main DTS Elements

Un *Node* en una DTS, especifica una caja contenedora de otros elementos, de la que aparecerán instancias en los modelos elaborados bajo la notación. Un *List Element* especifica (un tipo de) líneas de texto que usualmente formarán parte de una lista contenida en una instancia de un *Node*. Un *Connector* especifica una conexión en forma de línea entre instancias de *Nodes* u otros elementos. Un *Connector* es restringido por *Connection Constraints*, que aparecen conectando *Nodes* u otros elementos en la DTS, y especifican qué tipo de objetos pueden ser conectados con instancias del *Connector*.

Existen otros tres elementos que, en la DTS, expresan diferentes relaciones de contención para la notación gráfica: *Accessory part/whole*, *List Element part/whole* y *Node part/whole*. Un *Generic Specification* especifica una clase abstracta de la cual pueden heredar *Nodes*, *Connectors*, etc., pero que no puede ser instanciada directamente en la notación gráfica. Un *Basic Class* especifica un objeto sin representación gráfica, que puede ser usado como estructura de almacenamiento interno. *Nodes*, *Graphs*, *List Elements*, y otras especificaciones, pueden tener *Properties* y *Methods*. Un *Property* especifica un atributo de una instancia en la notación gráfica, que no aparece a la vista en el diagrama pero que se puede editar en un cuadro de diálogo de edición de propiedades. Otra manera de especificar un atributo es mediante un *Relationship* que se muestra, en la DTS, como una relación dirigida que conecta un elemento a otro, indicando el tipo del atributo y el contenedor del atributo. Un *Method* es un método programado en *Alter* (véase sección 4). Adicionalmente se pueden especificar menús, botones y enumeraciones (DOME 2004).

Con los elementos descritos es posible elaborar cualquier tipo de metamodelo en DOME y realizar las correspondientes instancias del mismo para la realización de los diferentes modelos de usuario, como se hace en cualquiera de las herramientas CASE. Es de especial interés en este artículo el metamodelo de UML expresado en MOF (Meta Object Facility), el formalismo para expresión de metamodelos de UML, tal como fue especificado por el OMG (OMG, 2004) y del cual en la Figura 2 se puede apreciar una versión simplificada tomada de Medvidovic et al. (2002).

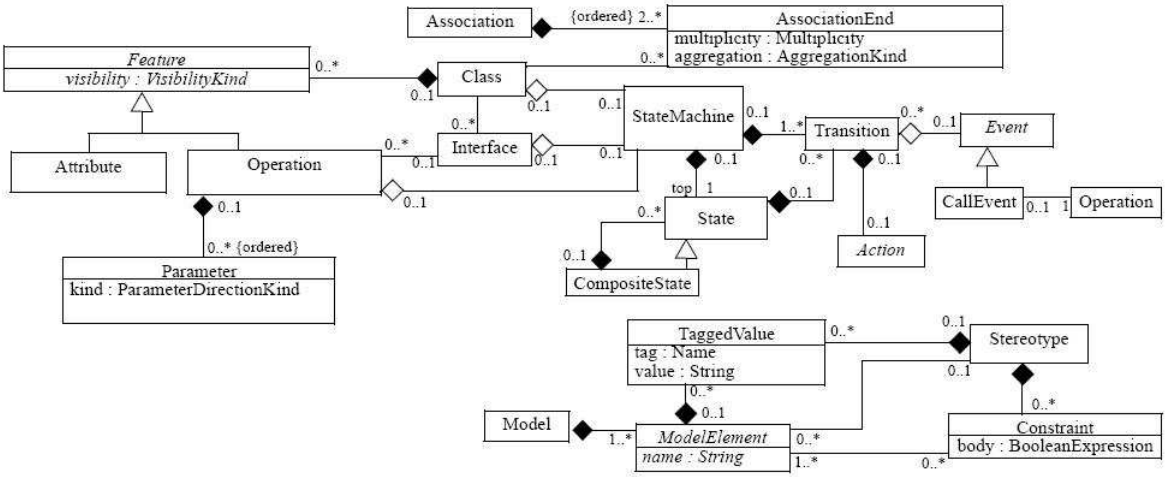

Figura 2. Metamodelo simplificado de UML en el formalismo MOF.
Figure 2. An UML simplified metamodel in the MOF formalism.

A modo de ejemplo, en la Figura 3 se observa parte de la DTS que especifica los aspectos estructurales de un diagrama de clases, como fue definida por el equipo creador de DOME (esta especificación está incluida en la versión 5.3 de DOME). En el medio de la Figura 3 aparece el *Node* *Class*[1] que hereda del *Generic Specification* *GeneralizableElement* de donde obtienen los *Atributes* *is-abstract*, *is-leaf* e *is-root*. Entre los *Methods* de *Class* aparecen tanto métodos de la versión original incluida en DOME (e.g. *qualified-name*), como algunos que fueron adicionados en el contexto de este trabajo, como *all-parents* y *are-distinguishable*. En la parte inferior la Figura 3, se pueden observar las relaciones de contención entre el *Node* *Class* y los elementos siguientes: el *List Element* *Atribute*; el *List Element* *Operation*; y el *Node* *ParameterBox*. También se puede observar abajo a la derecha de la Figura 3, el *Connector* *InheritsFrom* que es complementado por un *Connection Constraint* que aparece en la figura como una relación que sale y llega a *Class*, rotulada "*InheritsFrom*". Dicha *Connection Constraint* indica que una instancia del *Connector* *InheritsFrom*, en la notación gráfica, es una relación que une dos instancias de *Class*.

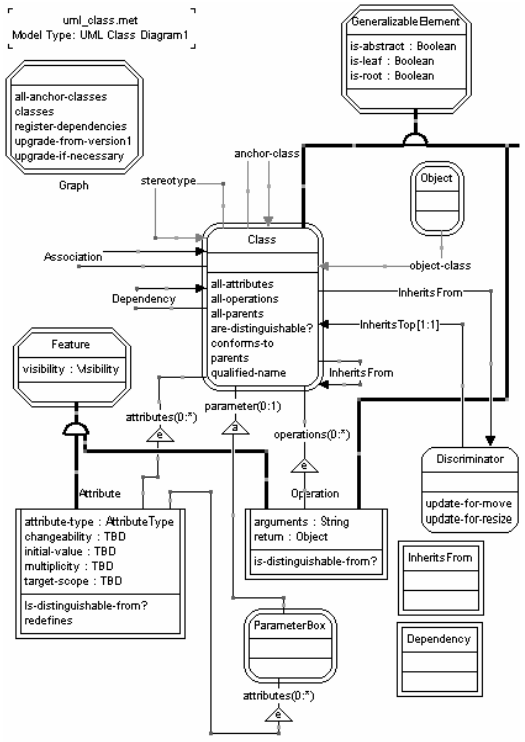

Figura 3. Parte de la DTS del diagrama de clases.
Figure 3. Portion of the class diagram DTS.

[1] Por claridad, los meta-metaelementos de DOME (e.g. *Node*) aparecen en cursiva mientras que los nombres de sus instancias (e.g. *Class*) aparecen en cursiva y subrayados.

A partir de la DTS descrita arriba, se genera un editor para diagramas de clases UML como el que se puede observar en la Figura 4. En la barra de herramientas ubicada en la parte izquierda aparecen íconos para *Note*, *Class*, *Attribute*, *Operation*, *Object*, *Parameter Box*, *Constraint*, *Association*, *InheritsFrom* y *Dependency*. Al lado derecho de la barra de herramientas, en la Figura 4, se observa la región de edición donde aparece un ejemplo de un diagrama de clases construido con el editor.

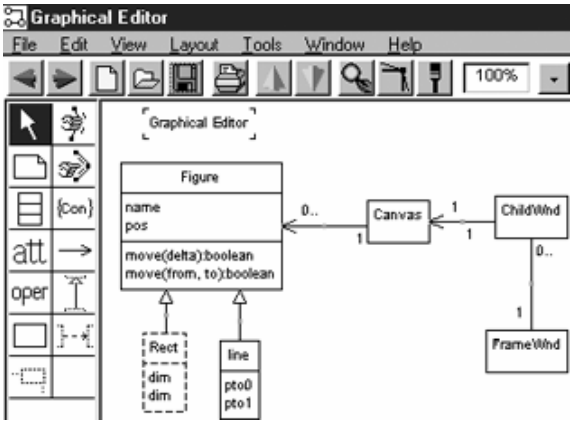

Figura 4. Ventana del editor de DOME para diagramas de clases UML, generado a partir de una DTS.

Figure 4. DOME Editor window for UML class diagram, generated from a DTS.

A más bajo nivel, se puede adicionar a la especificación de los aspectos estructurales, la especificación de otros aspectos de tipo lógico, tales como: restricciones más complejas, rutinas de análisis y generadores de código o documentación. Para ello, se debe usar el lenguaje de programación *Alter*, que es una implementación casi completa del lenguaje *Scheme*[2], con algunas extensiones que buscan realzar su funcionalidad como extensión de DOME. Aunque el propósito principal de *Alter*, es facilitar la transformación de modelos, complementando otra extensión grafica de DOME llamada Projector, también permite introducir en la DTS, reglas de consistencia que se verifiquen sobre diagramas construidos a partir de la notación gráfica definida por la DTS (Engstrom y Krueger 2000; Pereira 2001). En la sección 4 se extiende el análisis del lenguaje *Alter* y de los mecanismos para introducir reglas de consistencia.

[2] Cómo está definido en el documento Revised Report on the Algorithmic Language Scheme, por William Clinger, Jonathan Rees, et al., o por IEEE Std. 1178-1990, IEEE Standard for the Scheme Programming Language.

3. LA ESPECIFICACIÓN DE UML EN DOME

La versión 5.3 de DOME viene con una serie de DTS's construidas por el equipo de DOME. Entre ellas se encuentran varias asociadas con la notación UML, que incluyen diagramas de actividades, clases, colaboración, secuencias, estados y casos de uso. Estas DTS se tomaron como especificaciones de partida para expresar las reglas de consistencia de UML en DOME; por ello, antes de abordar el problema de la consistencia, se procederá a la realización de un análisis de las similitudes y discrepancias de la DTS del diagrama de clases en DOME en relación con la especificación de UML provista por el OMG (OMG, 2004). Una descripción más detallada de la DTS del diagrama de clases en DOME se puede obtener en el manual de DOME (DOME, 2004).

La especificación de UML en su versión 2.0 (OMG 2004) presenta diferencias fundamentales con la especificación DOME correspondiente. En primer lugar, la especificación de los aspectos estructurales de UML es expresada en el formalismo MOF (véase Figura 2 para consultar nuevamente el metamodelo simplificado), cuya apariencia es similar al diagrama de clases mismo (OMG 2004), mientras que DOME emplea su propio lenguaje de metamodelamiento. Una diferencia particularmente importante entre estos dos lenguajes, es que la especificación de los aspectos estructurales de una notación gráfica en DOME, incluye los aspectos gráficos propios de la notación definida.

Una consecuencia de la mezcla entre los aspectos estructurales y los gráficos se puede observar en la variedad de los elementos necesarios para la especificación del diagrama de clases de UML en DOME (véase Figura 3). Allí, *Class* es especificado por un

Node, *InheritsFrom* por un *Connector* y *Attribute* por un *List Element*. La especificación del OMG es completamente uniforme en su representación, puesto que *Class*, *Generalization* (que viene a ser el equivalente a *InheritsFrom* en la especificación de UML) y *Attribute* son especificados por medio de clases; la variedad de la representación de estos elementos en DOME es consecuencia de la necesidad de mezclar los aspectos estructurales con los aspectos gráficos del diagrama. Así, un *Node* sirve para representar elementos que aparecerán en la notación gráfica como cajas contenedoras de otros elementos (de allí que *Class* sea un *Node*), mientras que un *List Element* se usa para representar elementos que aparecerán en la notación gráfica contenidos por *Nodes* como parte de una lista (de allí que *Attribute* sea un *List Element*).

Pese a la dificultad para comparar la especificación del diagrama de clases inmerso en el documento del OMG con aquélla provista por el equipo de DOME, éstas comparten un lenguaje común y la estructura es similar. El lenguaje común se refiere a los nombres de los elementos modelados (tales como: *GeneralizableElement*, *Class*, *Association*, *AssociationEnd*, *Feature*, *Attribute*, *Operation*, *Dependency*) o sus atributos (i. e. *is-abstract*, *is-leaf* y *is-root* de *GeneralizableElement* o *is-navigable*, *aggregation* y *multiplicity* de *AssociationEnd*). Aunque hay algunas pequeñas diferencias en los nombres utilizados, en términos generales hay compatibilidad nominal.

En cuanto a la estructura, las relaciones entre los elementos de la especificación de UML[3], aunque sean expresadas de maneras diferentes, se preservan de la especificación del OMG a la DTS. Por ejemplo, la relación entre *Atributte* y *Class*, que en la especificación del OMG aparece implícitamente expresada por una relación de agregación entre *Classifier* (ancestro de *Class*) y *Feature* (ancestro de *Attribute*), en la DTS del diagrama de clases en DOME aparece representada mediante un *List Element part/whole* que conecta el *Node Class* al *List Element Attribute* (véase Figura 3).

Otro punto importante de discrepancia surge de la separación de UML en varios DTS's independientes para cada diagrama UML. En DOME, los elementos a nivel abstracto se especifican para servir a un diagrama específico. Por ejemplo, *Object* y *Message* aparecen como *Nodes* dos veces independientes en las DTS's del diagrama de colaboración y en la del diagrama de secuencia. Incluso el *Node Object* aparece también en la DTS del diagrama de clases. Esto además, imposibilita especificar un núcleo común a varios diagramas, lo que dificulta aun más basarse en la especificación del OMG.

Otra consecuencia de separar la especificación de UML en diagramas es la imposibilidad de imitar el uso de herencia que se observa en la especificación de la OMG. La falta de un núcleo común limita el surgimiento de generalizaciones de más alto nivel. Adicionalmente, la herencia en una DTS tiene ciertas limitaciones implícitas; debido a la variedad de elementos de modelamiento (*Node*, *List Element*, etc.) y sus particularidades, se ven limitadas las características generalizables.

4. EL LENGUAJE DE PROGRAMACIÓN *ALTER* Y LOS MECANISMOS PARA EXPRESAR CONSISTENCIA EN DOME

Como se mencionó en la sección 2, el lenguaje de programación *Alter* se puede utilizar para implementar reglas de consistencia más complejas que aquellas que se especifican gráficamente en una DTS. Para ello, en primer lugar, es posible incluir métodos codificados en *Alter*, sobre los *Nodes*, *Connectors*, *List Elements*, *Generic Specifications*, *Graphs* o *Basic Classes*. Un método se programa por ejemplo en un *Node*, escribiendo una expresión lambda que, en

[3] Una descripción más detallada de la especificación de UML en MOF que se muestra en la Figura 2 se puede obtener en los documentos de especificación del OMG (OMG, 2004).

tiempo de ejecución, ata el símbolo *self* a una instancia de dicho *Node* y, posiblemente, otros parámetros a otras instancias (DOME 2004). Ya con esta capacidad y el poder expresivo inherente a *Scheme*, utilizando un evaluador que posee DOME para sentencias de *Alter*, se podría hacer verificación de consistencia.

Sin embargo, hay otra característica de DOME, que permite verificar consistencia justo en los momentos en que puede ser violada, de modo que se prevenga o se advierta al modelador cuando cometa la falta. Esto se puede lograr programando en *Alter* ciertos métodos específicos sobre los *Nodes, Connectors, List Elements*, etc., que el editor generado a partir de la DTS, disparará como respuesta a ciertos eventos. Por ejemplo, a un *Node* llamado *primerNode* se le puede programar el método *Creation Check* que debe retornar un valor booleano; el método será ejecutado justo antes de instanciar un *primerNode* y, en caso de retornar *false* la creación será abortada, mientras que si el método retorna *true*, la instancia de *primerNode* será creada.

El lenguaje *Alter* presenta dos características que le facilitan servir en el sentido mencionado:

(1) Varias primitivas han sido adicionadas a aquellas especificadas en la versión 4 del Reporte revisado del Lenguaje *Scheme*, para soportar la manipulación de estructuras construidas con DOME.

(2) El *Alter* posee un conjunto amplio de primitivas para el manejo gráfico y basado en ventanas (DOME 2004).

En las secciones 5.1 y 5.2, a medida que se presente el código *Alter*, se introducirán algunas de las particularidades del lenguaje.

5. CASO DE ESTUDIO: DOS REGLAS DE CONSISTENCIA DEL METAMODELO DEL DIAGRAMA DE CLASES DE UML Y SU IMPLEMENTACIÓN EN DOME

Para el caso de estudio se seleccionaron dos reglas de consistencia de la especificación de UML tomadas de su versión 2.0[4] (véase Tabla 1). A continuación se presentan las reglas, aunque es necesario advertir que fueron modificadas de su versión original por varias razones: (1) Aquí las reglas se expresaron en términos de las metaclases instanciables del metamodelo de UML para poder hacer un paralelo a otra especificación que no incluye toda la estructura de herencia; (2) Una regla como la segunda sólo tiene sentido si se acompaña de cada una de las definiciones que ayudan a definirla y (3) se suprimieron algunos aspectos que no están modelados en la implementación provista por DOME. Las reglas son las siguientes:

1. Una clase no puede heredar directa ni indirectamente de si misma.
2. Todos los miembros (miembros heredados, atributos poseídos y operaciones poseídas) de una clase deben ser distinguibles dentro de ella.
   - Dos miembros **son distinguibles** dentro de una clase si (a) tienen tipos no relacionados o (b) si son ambos de tipo Operación, tienen diferente signatura, o (c) en caso de tener tipos relacionados (diferentes de operación), tienen nombre diferente.
   - Los **miembros heredados** de una clase *A*, son los miembros de las clases de las que hereda *A*, excepto aquellas propiedades que son redefinidas en *A*.
   - El concepto de **tipo relacionado** proviene del metamodelo. Dos instancias tienen tipos relacionados si son instancias de la misma metaclase o de metaclases que heredan una de la otra.
   - Dos Operaciones tienen **diferente signatura** si tienen diferente nombre, o en caso de tener el mismo nombre, la lista formada por el tipo de cada parámetro es distinta de una a la otra.

[4] Las reglas 1 y 2 aparecen prácticamente inmodificadas en la versión 1.5 de la especificación de UML.

En las siguientes dos secciones se detalla la implementación en DOME de las reglas de consistencia mencionadas, partiendo de la especificación en OCL entregada por el OMG (OMG 2004). Finalmente en la sección 5.3 se hace un análisis de la implementación realizada.

Tabla 1. Referencias a la especificación de UML donde aparecen las reglas a implementar (OMG, 2004).
Table 1. UML specification references that show the rules to implement (OMG, 2004)

Regla	Sección UML2.0	Restric-ciones	Operaciones Adicionales
1	7.8.1 classifier	2	2, 3
2	7.3.3 Namespace	1, 2	1, 2
	7.3.2 NamedElement		2
	7.9.1 BehavioralFeature		1
	7.13.1 Package		2
	7.8.1 Classifier	4	4, 5
	7.11.3 Class		1

5.1. REGLA 1: HERENCIA CIRCULAR

En la especificación de UML 2.0 la regla en cuestión se expresa mediante la siguiente expresión OCL en el contexto de *Classifier*:

```
not self.allParents()->includes(self)
```

Además, la función *allParents* no es una primitiva del lenguaje sino que, en el contexto de *Classifier*, se especifica así:

```
allParents =
  self.parents()->
    union(self.parents()->collect(p |
      p.allParents())
```

y, por último, la función *parents* se especifica de la siguiente forma en el contexto de *Classifier*:

```
parents = generalization.general
```

En DOME se trata de hacer algo similar. Se implementó un método *parents* en el *Node Class*, utilizando el lenguaje *Alter*, así:

```
(add-method (parents (UMLClass) self)
  (map destination
    (select (outgoing-arcs self)
      (lambda (x) (is-a? x
UMLInheritsFrom)))))
```

Donde la primitiva *outgoing-arcs*[5] retorna una lista con todas las conexiones que salen de su único argumento (instancia de un *Node*). La primitiva *select* recibe dos argumentos, una lista *list* y un procedimiento de un argumento *proc*, y retorna una lista con los elementos de *list* para los cuales *proc* retorna *true*. Finalmente la primitiva *is-a?* recibe un objeto y el nombre de un tipo y retorna *true* si el objeto es del tipo indicado o *false* de lo contrario.
Luego, con base en este método, se implementa el método *all-parents* así:

```
(add-method (all-parents (UMLClass) self)
  (let ((p (parents self)))
    (if (eq? p '())
      '()
      (append p
        (apply append (map all-parents
p)))))))
```

La manera de implementar la regla en sí, depende de la manera en que se quiera verificar. En este trabajo la elección fue programar el método *Creation Check* del *Connector* *InheritsFrom*, de manera que, antes de instanciar una relación de herencia, el editor verifique si dicha inserción provocaría una herencia circular y en tal caso no se permita su instanciación. El método queda implementado así:

```
(lambda (container . args)
  (not (memq (car args)
    (all-parents (cadr args)))))
```

Donde la lista *args*, tiene como primer elemento el origen de la herencia y como segundo elemento el destino de la herencia.

[5] En adelante tan solo se explicarán las particularidades del lenguaje *Alter*, omitiendo de la discusión todo lo que *Alter* tiene en común con *Scheme*.

5.2. REGLA 2: UNICIDAD DE NOMBRES

En la especificación de UML2.0 la regla en sí es condensada en la invariante *membersAreDistinguishable()* en el contexto de *NameSpace*, lo cual se complementa con la definición de la función:

```
Namespace::membersAreDistinguishable() :
Boolean;
membersAreDistinguishable =
  self.member->forAll( memb |
    self.member->excluding(memb)-
>forAll(other |
      memb.isDistinguishableFrom(other,
self)))
```

Aunque el conjunto *self.member* queda implícitamente definido por el metamodelo de UML y la semántica de OCL, la restricción 4 (véase Tabla 1) de *Classifier* complementa la definición al especificar inheritedMember que, para el caso de *Class*, retorna un subconjunto de *member*. De la misma manera, la restricción 2 (véase Tabla 1) de *Namespace* especifica *importedMember* que en el caso de una *Class*, retorna un subconjunto de *member*.

Más importante es la definición de la función *isDistinguishableFrom*, definida en el contexto de *NamedElement*

```
NamedElement::isDistinguishableFrom(n:Na
medElement, ns: Namespace): Boolean;
isDistinguishable =
  if self.oclIsKindOf(n.oclType) or
    n.oclIsKindOf(self.oclType)
  then ns.getNamesOfMember(self)->
    intersection(ns.getNamesOfMember(n))-
>
      isEmpty()
  else true
endif
```

Y su redefinición en *BehavioralFeature*, que busca tener en cuenta toda la signatura

```
BehavioralFeature::isDistinguishableFrom(n
: NamedElement, ns: Namespace): Boolean;
isDistinguishableFrom =
  if n.oclIsKindOf(BehavioralFeature)
  then
    if ns.getNamesOfMember(self)->

  intersection(ns.getNamesOfMember(n))->
      notEmpty()
    then Set{}->including(self)-
>including(n)->
      isUnique( bf | bf.parameter->
        collect(type))
    else true
    endif
  else true
endif
```

Cabe anotar que parte de la complejidad de las dos últimas piezas de código OCL, es insustancial para el análisis que se realiza debido a que, en el modelo de clases de DOME modificado para este trabajo, un miembro tiene un único nombre, mientras que el metamodelo de UML permite que un mismo miembro tenga varios nombres, producto de varias importaciones bajo diferentes alias. Por tanto la expresión

```
ns.getNamesOfMember(self)->
  intersection(ns.getNamesOfMember(n))->
    notEmpty()
```

podría ser reemplazada, para este caso, por self.name = n.name.

De nuevo, en DOME se trata de imitar la estructura, utilizando *Alter*. Se agregan métodos *is-distinguishable-from?* al *List Element* *Attribute* y al *List Element* *Operation*, especificados respectivamente por las siguientes expresiones lambda.

```
(add-method (is-distinguishable-from ?
(UMLAttribute) self other)
  (not (equal? (name self) (name other))))

(add-method (is-distinguishable-from ?
(UMLOperation) self other)
  (or (not (equal? (name self) (name other)))
      (not (equal? (length (get-property
"arguments" self))
```

```
      (length (get-property "arguments" other))))
      (not (equal? (length (get-property "arguments" self)) 0))
      (not (apply and (map equal? (get-property "arguments" self)
        (get-property "arguments" other))))))
```

Donde la primera expresión imita el genérico *NamedElement::isDistinguishableFrom* y la segunda, definida para el *List Element* Operation, hace el papel de la versión redefinida en *BehavioralFeature*.
En la anterior pieza de código se introdujo la primitiva de *Alter get-property*. Este método recibe dos argumentos, una cadena de caracteres *propName* y un objeto object que pueda poseer *Properties* como una instancia de un *Node* o de un *List Element*. Retorna el valor de la *property* identificada con el nombre *propName* de *object*. También se introdujo la primitiva *name*, que recibe un objeto y retorna su nombre (el objeto debe poseer nombre).

A continuación, fue necesario agregar los métodos *all-attributes* y *all-operations* al *Node* Class, para recolectar los atributos (y operaciones respectivamente) heredados, teniendo en cuenta no repetir aquellos atributos (u operaciones) que son redefiniciones de atributos declarados más arriba en la descendencia. A continuación se presenta su codificación en *Alter*:

```
(add-method (all-attributes (UMLClass) self)
  (letrec ((attsSelf (select (elements self) (lambda (x) (is-a? x UMLAttribute))))
           (parentsSelf (parents self))
           (attsParents (if (eq? parentsSelf '()) '() (apply append (map all-attributes parentsSelf))))
           (appendNoRepeat (lambda (l1 l2)
             (cond ((eq? l1 '()) l2)
               ((detect l2 (lambda (x) (not
                   (is-distinguishable-from? x (car l1)))) #f)
                 (appendNoRepeat (cdr l1) l2))
               (else (cons (car l1)
                           (appendNoRepeat (cdr l1) l2)))))))
    (appendNoRepeat attsParents attsSelf)))


(add-method (all-operations (UMLClass) self)
  (letrec ((opsSelf (select (elements self) (lambda (x) (is-a? x UMLOperation))))
           (parentsSelf (parents self))
           (opsParents (if (eq? parentsSelf '()) '() (apply append (map all-operations parentsSelf))))
           (appendNoRepeat (lambda (l1 l2)
             (cond ((eq? l1 '()) l2)
               ((detect l2 (lambda (x) (not
                   (is-distinguishable-from? x (car l1)))) #f)
                 (appendNoRepeat (cdr l1) l2))
               (else (cons (car l1)
                           (appendNoRepeat (cdr l1) l2)))))))
    (appendNoRepeat opsParents opsSelf)))
```

La primitiva de *Alter detect* es similar a *select* reseñada en la sub-sección anterior, excepto que sólo retorna el primer elemento de la lista que satisfaga el predicado (que retorne *true* al procedimiento).
Finalmente se escribió una función genérica que comprueba si los elementos de una lista de miembros de una *class* son distinguibles.

```
add-method (are-distinguishable? (UMLClass) self members)
  (do ((attributes members (cdr attributes)))
    ((or (eq? attributes '()) (eq? (cdr attributes) '())
      (not (apply and (map
          (lambda (x) (is-distinguishable-from? (car attributes) x))
          (cdr attributes)))))
    (or (eq? attributes '()) (eq? (cdr attributes) '())))))
```

Con lo que la regla de consistencia queda expresada mediante la siguiente expresión:

```
(and    (are-distinguishable?
          self (all-attributes self))
        (are-distinguishable?
```

self (all-operations self)))

Esta expresión se incluyó en el método *Line Style* del *Node* *Class* de la siguiente manera:

```
(lambda (self)
  (if (and   (are-distinguishable? self (all-
attributes self))
             (are-distinguishable? self (all-
operations self)))
      'normal
      'simpledash))
```

El método *Line Style* es ejecutado por el editor generado cada vez que una instancia del *Node* requiere ser repintada, y retorna un estilo de línea que se emplea para delinear la instancia del *Node*. Lo que se logra entonces con el código presentado es que cuando una *class* tiene miembros no distinguibles, su borde sea delineado con una línea punteada, mientras que si todos sus miembros son distinguibles entre sí, la *class* es delineada por una línea sólida. En la Figura 4, se observa cómo la clase *Rect* posee dos atributos no distinguibles (*dim*) y se puede apreciar el borde de la clase que aparece punteado.

5.3 ANÁLISIS DE LA IMPLEMENTACIÓN

En primer lugar, el lenguaje *Alter*, combinado con la descripción orientada a objetos de la DTS, ofrece una flexibilidad muy amplia en cuanto a verificación de consistencia. Prueba de ello es la implementación del método *all-operations* que tiene en cuenta no sólo la signatura de las Operations sino también la redefinición. Además, los mecanismos ofrecidos para verificar la consistencia en tiempo real, son suficientemente efectivos para desarrollar un entorno de edición confortable para una notación gráfica.

Sin embargo, la legibilidad del código *Alter* es baja. Esto se debe en parte a la inherente ilegibilidad de la sintaxis prefija de *Lisp*, pero también a la manera en que se accede a datos del modelo con *Alter*. La implementación del método *parents*, presentada en la sección 5.1, ilustra este hecho claramente: mientras en OCL una sola línea de código basta para implementarlo, en *Alter* hace falta seleccionar (expresión *select*) de entre las relaciones que parten de la clase, aquellas de tipo *InheritsFrom;* Adicionalmente, es necesario aplicar el método *destination* a cada una de las relaciones (expresión *map*) para obtener el extremo opuesto de cada una. A medida que crece la complejidad, crece aún más este problema como se observa en el método *all-attributes*, donde se requiere definir un método interno *appendNoRepeat* para seleccionar todos los atributos sin que se repitan las redefiniciones, ocultando el sentido declarativo de la regla.

Finalmente, es importante destacar que la agrupación del código que fue presentada en las secciones 5.1 y 5.2, es producto de la recolección de cada una de las piezas de código dispersas a través de toda la especificación. Al enfrentarse directamente con la implementación, surgen varios problemas en la búsqueda de dicho código, cuyas piezas se encuentran esparcidas por la DTS. Sin embargo, el mismo problema surge al intentar compendiar las reglas de consistencia de la especificación de UML provista por el OMG. El esfuerzo por separar el código en pequeñas partes coherentes en sí mismas y referentes al contexto adecuado, causa la dispersión del total de la información concerniente a una regla en sí.

6. CONCLUSIONES Y TRABAJOS FUTUROS

Se presentó en este artículo la implementación de dos reglas de consistencia de la especificación de UML 2.0 (OMG 2004), para un editor generado con la herramienta metaCASE DOME. Si bien la complejidad se incrementó al emplear el lenguaje de programación *Alter*, en comparación con la especificación de las reglas en OCL, fue posible concebir una implementación coherente.

Alter hereda el poder expresivo de los lenguajes funcionales tipo *Lisp*, pero dificulta preservar el sentido declarativo de las reglas de consistencia. Por esa misma naturaleza

diferente de *Alter* frente a OCL, la tarea de traducir las reglas escritas en OCL (OMG 2004) a *Alter* es dispendiosa. Pero, además, la traducción se dificulta aún más, debido a las diferencias inevitables entre la especificación de los aspectos estructurales de UML al pasar del documento de la OMG (OMG 2004) a DOME.

Por otro lado, la manera en que el usuario del editor percibe el control sobre las reglas, es aceptable, gracias a los mecanismos que ofrece DOME para programar la respuesta a algunos eventos del editor. Sin embargo, el conjunto de eventos programables es reducido lo que dificulta la programación.

Finalmente, para la continuación de este trabajo, se espera introducir un conjunto más nutrido de reglas de consistencia, sobre un editor de UML generado con DOME. Vale la pena abordar también otros diagramas, además del de clases, pero la necesidad de separar las especificaciones de los diferentes diagramas de UML en DOME dificultará introducir reglas de consistencia que involucren varios diagramas.

REFERENCIAS

[1] OMG. *OMG Unified Modeling Language Specification.* Object Management Group. Available: http://www.omg.org/UML/. [Citado 8 de Diciembre de 2004].

[2] ENGSTROM, E. y KRUEGER, J. 2000. *Building and rapidly evolving domain-specific tools with DoME.* Proceedings of the IEEE International Symposium on Computer-Aided Control System Design, 2000. CACSD 2000. Pp. 83-88.

[3] DOME. *What is Dome.* Available: http://www.htc.honeywell.com/dome/description.htm. [Citado 8 de Diciembre de 2004].

[4] DE LARA, J., y VANGHELUWE, H.. *AToM^3: A tool for Multi-Formalism and Meta-Modelling*. Proceedings of the Fifth International Conference on Fundamental Approaches to Software Engineering. Pp. 174-188. 2002.

[5] METAEDIT+. MetaCase Consulting. Available: http://www.metacase.com. [Citado 8 de Diciembre de 2004].

[6] METAGME (Meta-Graphical Model Editor). Vanderbilt University Institute for Software Integrated Systems (ISIS). Available: http://www.isis.vanderbilt.edu/Projects/gme/default.html [Citado 8 de Diciembre de 2004].

[7] DIAGEN. University of Erlangen. Available: http://www2.informatik.uni-erlangen.de/DiaGen/ [Citado 8 de Diciembre de 2004].

[8] CODESIGN. *The CodeSign Project*. Available: http://www.tik.ee.ethz.ch/~codesign/ [Citado 8 de Diciembre de 2004].

[9] R. ESSER and J. W. JANNECK. *A Framework for Defining Domain-Specific Visual Languages*. Workshop on Domain Specific Visual Languages, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA-2001). 2001.

[10] MEDVIDOVIC, N., ROSENBLUM, D., REDMILES, D. y ROBBINS, J. Modeling software architectures in the Unified Modeling Language. En: ACM Transactions on Software Engineering and Methodology (TOSEM), Volumen 11, No. 1, 2002. Pp. 2-57.

[11] M. A. PEREIRA REMELHE. *Simulation and VisualizationSupport for User-defined Formalisms Using Meta-Modeling and Hierarchical Formalism Transformation*. Proceedings of the 2001 IEEE International Conference on Control Applications. México City. 2001.