UN-LEND: Un lenguaje para la especificación de modelos de UN-MetaCASE

Carlos Mario Zapata y Fernando Arango

UNIVERSIDAD NACIONAL DE COLOMBIA. Facultad de Minas.
Escuela de Sistemas. Grupo UN-INFO
{cmzapata ; farango}@unalmed.edu.co

Recibido para revisión Jul 2004, aceptado Sep 2004, versión final recibida Sep 2004

Resumen: Las herramientas CASE han contribuido a la construcción de modelos que representan diferentes aspectos de un problema con miras a su traducción en una pieza de software; sin embargo, estas herramientas poseen una capacidad limitada en la cantidad de tipos de modelos que permiten elaborar. Desde hace algunos años han venido surgiendo herramientas MetaCASE, como AToM^3 y DOME, en las cuales los modelos no están definidos sino que se construyen en un formalismo de tipo gráfico o textual apoyado en un lenguaje para la expresión de la sintaxis del modelo y su forma de graficación. En este artículo se presenta UN-LEND, el lenguaje creado por el grupo UN-INFO para la especificación de modelos del UN-MetaCASE que separa la lógica del modelo de su representación gráfica, con el fin de facilitar la utilización de múltiples visualizaciones del mismo modelo.

Palabras Clave: Metamodelamiento, Lenguajes de especificación de modelos

Abstract: CASE tools have contributed to model building for representing several aspects of a problem, for converting it in source code of a software piece; however, these tools have limited capacity in the amount of type models they can elaborate. Some years ago researchers have been developing MetaCASE tools, e.g. AToM^3 and DOME. Models in this kind of tools aren't defined; instead of, models are built in a graphic or textual formalism, supported in a language for modeling syntax expression and graphic representation. In this paper, we present UN-LEND, a language created by UN-INFO research group for UN-MetaCASE model specification; this language torn apart model logic from graphic representation, for facilitating multiple view usage of the same model.

Keywords: Metamodeling, Model Specification Languages

1 INTRODUCCIÓN

Con el surgimiento del UML a mediados de la década de 1990 [OMG (2004)] el diseño del software obtuvo una gran cantidad de modelos que contribuyeron a explicar el comportamiento y la estructura subyacente en las piezas de software antes de siquiera acometer su construcción. En esta época surgieron también muchas de las piezas de software denominadas CASE (Computer-Aided Software Engineering) que brindan apoyo a los analistas y diseñadores del software en la construcción de los modelos, suministrándoles entornos gráficos amigables y facilidades para la elaboración y edición de modelos predefinidos mediante plantillas, que incluyen las componentes básicas de los principales modelos, especialmente de UML.

El surgimiento y utilización de otros modelos diferentes a UML en las diferentes etapas de desarrollo de software motivó la aparición de otras herramientas CASE que no se limitaran a los diagramas convencionales, sino que tuvieran la posibilidad de definir cualquier tipo de modelos a través de su parametrización; esas herramientas, comúnmente denominadas MetaCASE, permiten la especificación de los diferentes modelos, para luego permitir su uso en la generación de instancias particulares, como si fueran herramientas CASE convencionales. Para realizar esta labor, estas herramientas necesitan un mecanismo para la representación de la estructura de los modelos, que generalmente es de tipo gráfico, pero que internamente es definido a través de un lenguaje que especifica la representación gráfica del metamodelo y su estructura lógica. Este tipo de especificación representa un obstáculo si se desea realizar la transformación de un modelo a otro, o cambiar la apariencia del modelo, puesto que sería necesario reescribir la especificación del metamodelo para modificar ya sea la

apariencia o la estructura.

UN-MetaCASE es una herramienta para el apoyo de los trabajos de modelamiento que se encuentra actualmente en desarrollo en la Escuela de Sistemas de la Universidad Nacional; en ella se desarrolló un lenguaje de especificación de metamodelos denominado UN-LEND, que permite una solución al problema de la combinación de la representación gráfica y la estructura lógica de los modelos. En UN-LEND se representa únicamente la lógica del metamodelo, pero puede interactuar con el entorno gráfico del UN-MetaCASE para definir la representación gráfica de los modelos, permitiendo la funcionalidad de las herramientas MetaCASE convencionales, pero con la flexibilidad que suministra la independencia de la especificación gráfica de la estructura lógica. En este artículo se describe el UN-LEND y se muestra la herramienta de apoyo para su interpretación, como punto de partida para la expresión posterior de modelos gráficos en UN-MetaCASE.

El artículo está organizado así: en la Sección 2 se presentan algunas generalidades en relación con el modelamiento mediante herramientas CASE y el metamodelamiento apoyado en herramientas MetaCASE; el formalismo de representación de AToM^3 y DOME se discute en la Sección 3; el UN-LEND y su notación se introducen en la Sección 4, al igual que se presenta el analizador sintáctico que ayuda a su interpretación; finalmente, las Secciones 5 y 6 presentan las principales conclusiones y trabajos futuros.

2 HERRAMIENTAS CASE Y METACASE

En la década de los setenta se comenzó a pensar en la posibilidad de apoyar los procesos de la recién creada Ingeniería del Software (IS) con herramientas computacionales [Bubenko, Langerfors y Solvberg (1971)]. El Dr. John Manley, director del Instituto de Ingeniería del Software de la Universidad de Carnegie Mellon (SEI, por sus siglas en inglés) a principios de la década de los ochenta, fue presumiblemente la primera persona en acuñar el término Computer-Aided Software Engineering [Burkhard y Jenster (1989)], para referirse a un grupo de procesos, técnicas y herramientas que apoyan la realización de diferentes procesos de IS, desde el modelamiento hasta la construcción, incluyendo el mantenimiento y la reingeniería. Las herramientas CASE alcanzaron su esplendor hasta mediados de la década de los noventa, cuando el surgimiento del Lenguaje Unificado de Modelamiento (UML por sus siglas en inglés) hizo necesaria la construcción de una serie de modelos conceptuales para representar la solución a problemas específicos del mundo [OMG (2004)]; entonces surgieron muchas herramientas con la función específica de permitir la elaboración de modelos típicos de UML, como el diagrama de clases, los diagramas de colaboración y secuencias, los diagramas de casos de uso y los diagramas de transición de estados, por mencionar sólo algunos. Una de esas herramientas, denominada ArgoUML®, que se aprecia en la Figura 1, permite la traducción a diferentes lenguajes de codificación, así como el manejo de algunas restricciones en los modelos.

Sin embargo, los diagramas que se pueden obtener con las diferentes herramientas CASE disponibles son limitados (por ejemplo, ArgoUML® únicamente soporta la realización de siete de ellos) y no se pueden crear otros nuevos; adicionalmente, si bien soportan algunas reglas de consistencia, no se pueden añadir otras nuevas, lo que se constituye en un esquema bastante rígido que impide la adición de nuevos elementos o depuración de los existentes. Esta rigidez de las herramientas CASE contrasta con el dinamismo de los lenguajes de modelamiento (UML en este caso), que permanentemente están liberando nuevas versiones que no pueden ser incorporadas en estas herramientas; además, UML no es el único lenguaje estándar de modelamiento, pues existen modelos y formalismos diferentes de UML que podrían contribuir a la definición y especificación adecuada de la solución a un problema en IS (tales como el diagrama causa - efecto, el diagrama de procesos y el diagrama de objetivos).

La solución a este problema surgió cuando se procuró la elaboración de un lenguaje para describir el lenguaje mismo en que estaban escritos los modelos de las herramientas CASE [Vangheluwe, De Lara y Mosterman (2002)], un esfuerzo incluso iniciado por el mismo UML mediante un formalismo denominado MOF (Meta Object Facility), que es muy similar al UML que trata de representar [OMG (2004)]. Sin embargo, los formalismos gráficos todavía son bastante cercanos a los modelos que representan, lo que finalmente impide la expresión de ciertas restricciones que deben ser cumplidas por los modelos que se instancian a partir del metamodelo. La solución para el lenguaje estándar UML fue la introducción de OCL (Object Constraint Language), un lenguaje que puede expresar más fácilmente ese tipo de restricciones; sin embargo, ni UML ni OCL son lenguajes ejecutables. Esa deficiencia la han subsanado herramientas metaCASE como el GME [Ledeczi, Maroti, Bakay, Karsai, Garrett, Thomason IV, Nordstrom, Sprinkle y Volgyesi (2001)] elaborando subconjuntos de UML y de OCL que se pueden ejecutar para realizar ciertas verificaciones de los modelos; sin embargo, la potencia expresiva del GME en comparación con el UML y el OCL originales aún es baja.

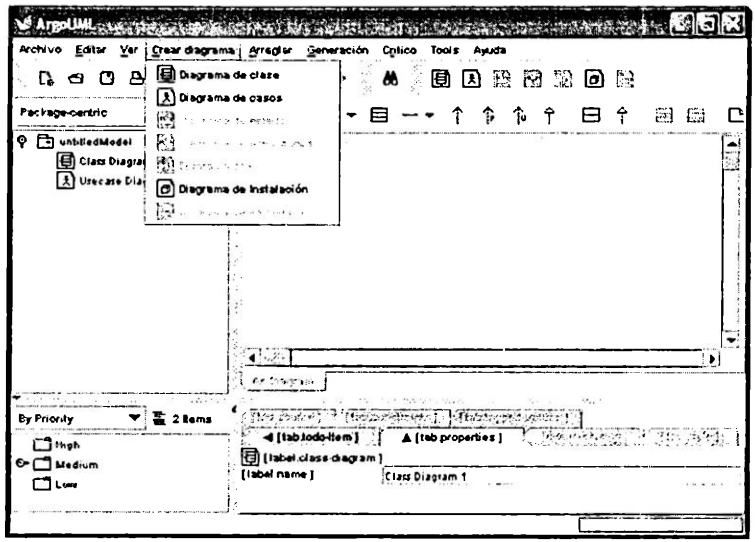


Figura 1: Imagen de ArgoUML® una herramienta CASE basada en diagramas UML

Otras herramientas MetaCASE en lugar de utilizar UML y OCL como lenguajes formales para la representación estructural de modelos, fueron creando sus estándares gráficos apoyados por un lenguaje propio de metamodelamiento. Entre estas herramientas se destacan DOME y AToM^3, las cuales tienen características que las hacen importantes en el ámbito de los MetaCASEs. DOME, por ejemplo, se caracteriza por su madurez, el Lenguaje lógico de especificación que utiliza y poseer implementación de modelos UML. AToM^3 se caracteriza por su orientación a la transformación de modelos por medio de gramática de grafos y por su código abierto (Python), que posibilita la adición de características especiales por parte del usuario. Dadas las características anotadas para estas dos herramientas metaCASE, sus formalismos se describen en la sección siguiente.

3 FORMALISMOS PARA METACASES ATOM^3 Y DOME

Dentro de las herramientas MetaCASE disponibles se presentan algunas que utilizan UML como formalismo para representación de los diferentes metamodelos, de manera similar a como lo realizan las herramientas CASE convencionales; otras herramientas emplean formalismos propios y diferentes que se apoyan más en lenguajes formales, si bien utilizan un formalismo gráfico para facilidad de realización de los metamodelos. En esta sección se describen los lenguajes formales empleados para la expresión de los metamodelos de AToM^3 [De Lara y Vangheluwe (2002)] y [DOME (2004)], las cuales poseen editores gráficos que ayudan al usuario a la creación e instanciación de modelos. Adicionalmente, se discuten las dificultades inherentes a los lenguajes formales que apoyan los metamodelos gráficos.

3.1 AToM^3:A Tool for Multi-Formalism Modelling and Meta-Modelling

Es una herramienta MetaCASE que utiliza un formalismo gráfico basado en el modelo entidad-relación para la expresión de los metamodelos [De Lara y Vangheluwe (2002)], el cual se soporta internamente en código fuente escrito en el lenguaje de programación Python [PYTHON (2004)]. En la Figura 2 se presenta la expresión de un pequeño subconjunto del metamodelo de clases expresado en el lenguaje entidad-relación de AToM^3.

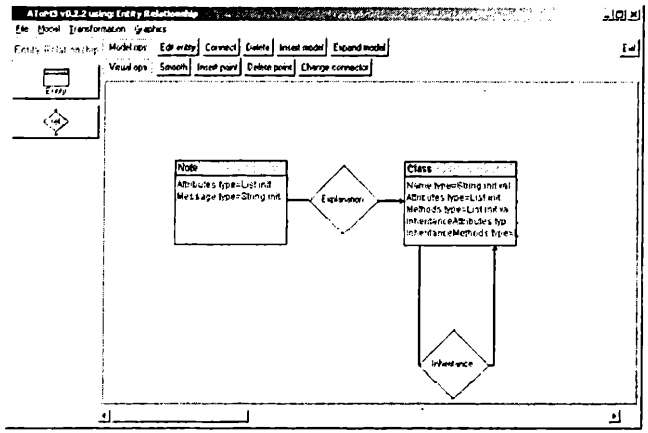


Figura 2: Metamodelo reducido del diagrama de clases expresado en el modelo E-R de AToM^3

Una vez se ha elaborado la expresión gráfica del metamodelo, se genera el código python que apoya el proceso de utilización del metamodelo en la herramienta. En AToM^3 se generan tres archivos que recogen tres aspectos diferentes del metamodelo:

1. Un archivo que incluye la expresión del metamodelo en términos del modelo entidad-relación, al cual se le coloca al final del nombre la cláusula "_ER", para diferenciarlo de los demás archivos (Por ejemplo "ClassModel_ER.py", siendo *py* la extensión de los archivos python). En este archivo se realiza la definición de los elementos con sus atributos lógicos y gráficos y las restricciones que sean aplicables. En la Figura 3 se aprecia un fragmento del código anotado; en ella, los puntos suspensivos indican partes del código fuente que se omiten por simplicidad, en (1) se muestra la definición de la apariencia de una entidad "Class", en (2) se define una restricción para la misma entidad y en (3) se muestra información de la posición de ese elemento en particular. Las porciones (1) y (3) del código definen características gráficas que deberían ser independientes del metamodelo, en tanto que (2) es propiedad de la lógica del metamodelo.

2. Un archivo que representa el formalismo del metamodelo, al cual se le coloca en el nombre la cláusula "_MM" (Por ejemplo ClassModel_MM.py). En este archivo se describe la manera como debe proceder de manera lógica AToM^3 cuando se realice la creación de una instancia particular del metamodelo. En la Figura 4 se muestra una porción del código incluido en este archivo; en la parte (1) de la Figura se muestran las instrucciones necesarias para la definición de los elementos que se pueden usar en la instanciación del metamodelo, en la parte (2) se definen los lementos con los cuales se puede conectar una instancia de la entidad Class, en la parte (3) se verifica el cumplimiento de las restricciones y en la parte (4) se definen las coordenadas en la pantalla. También en este caso hay elementos que pertenecen a la lógica del metamodelo (partes 1 a 3) y una parte (4) que únicamente atiende la representación.

3. Un archivo que se encarga de ejecutar el formalismo, al cual no se le coloca ninguna cláusula para identificarlo (por ejemplo ClassModel.py). Este archivo utiliza la definición del metamodelo y se encarga de controlar la ejecución de las restricciones. En la Figura 5 se presenta una parte del código correspondiente; en la parte (1) se muestra la instrucción que carga el archivo del metamodelo, en la parte (2) se definen las características gráficas de la entidad clase, en la parte (3) se define un archivo gráfico para el botón que controla la creación de las clases en el modelo y en la parte (4) se definen las diferentes acciones que se van a realizar cuando se realice la creación de una clase. Como en los casos anteriores, existen elementos correspondientes a la parte gráfica del modelo (partes 2 y 3) y elementos que describen la lógica del modelo (parte 4); adicionalmente aparece un elemento de comunicación entre los archivos (parte 1).

```
def ClassModel_ER(self, rootNode)

  self.obj20.appearance.setValue( ('Class', self.obj20))
  self.obj20.name.setValue('Class')                                (1)

                                                                   (2)
  cobj2=ATOM3Constraint()
  cobj2.setValue(('HerenciaCiclica', (['Python', 'OCL'], 1), (['PREcondition', 'POSTcondition'], 1), (['EDIT', 'SAVE', 'CREATE',
'CONNECT', 'DELETE', 'DISCONNECT', 'TRANSFORM', 'SELECT', 'DRAG', 'DROP', 'MOVE'], [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]), 'def
filtrar(x): return x.__class__.__name__=="Inheritance"\nfor obj1 in self.parent.ASGroot.listNodes[\'Class\']           #devuelve
todos los \'Class\'\n    obj2 = obj1\n    clases = [obj2]\n    pos = len(clases)\n    while len(filter(filtrar, obj2.out_connections_)) > 0:\n
for obj3 in filter(filtrar, obj2.out_connections_)                        #devuelve todos los \'Inheritance\' de los \'Class\' anteriores\n
if(pos != len(clases)):   \n                     clases = clases[pos]\n                          for obj4 in obj3.out_connections_
#devuelve todos los \'Class\' conectados a los \'Inheritance\' anteriores\n           if(obj4 in clases):\n                ciclo = "["\n
for i in (clases[clases.index(obj4):] + [obj4]):\n                 ciclo = ciclo + i.Name.toString() +", "\n               ciclo =
ciclo[:len(ciclo)-2] +"]"\n                  return("Herencia Ciclica no permitida " + ciclo, self)\n                   else
clases.append(obj4)\n                    obj2   =   obj4\n                                        pos   =   len(clases)\nreturn
None\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n'))
  lcobj2.append(cobj2)

  if self.genGraphics:
    from graph_EREntity import *
  ( new_obj = graph_EREntity(391.0,102.0,self.obj20) )             (3)
    new_obj.DrawObject(self.UMLmodel)
    self.UMLmodel.addtag_withtag("EREntity", new_obj.tag)
  else: new_obj = None

newfunction = ClassModel_ER
loadedMMName = 'EntityRelationship'
```

Figura 3: Parte del código python incluido en el archivo ClassModel_ER.py

Como se discutió a partir de las Figuras 3, 4 y 5, la definición del modelo en el formalismo requiere la combinación de características lógicas y gráficas, en la cuales se combina lo que quiere representar la parte lógica del modelo con las características de su representación visual. Además, se atomiza la funcionalidad del metamodelo y su distribución en tres archivos diferentes (e incluso más, puesto que la creación de un modelo en AToM^3 genera otros archivos adicionales de código python para la creación de la representación de cada dibujo en pantalla o para la ejecución de cada restricción en la creación de la instancia del metamodelo).

3.2 DOME: Domain Modeling Environment

La organización Honeywell creó esta herramienta MetaCASE que emplea un formalismo gráfico basado en el diagrama de clases de UML [OMG (2004)] pero adicionando diferenciaciones para las clases y las conexiones debido a la necesidad de incluir diferenciaciones para las representaciones gráficas [DOME (2004)]. Internamente, el formalismo se apoya en el lenguaje Smalltalk [SMALLTALK (2004)] combinado con Alter [DOME (2004)], un lenguaje propio de la especificación de DOME que es similar al Lisp y que utilizan para la codificación de métodos y restricciones [DOME (2004)]. En la Figura 6 se aprecia el modelo de clases expresado en el formalismo gráfico de DOME.

A diferencia de AToM^3, DOME apoya el formalismo gráfico en un solo archivo que contiene las especificaciones gráficas y lógicas del metamodelo, al cual se le coloca la extensión .met; ese archivo incluye la representación lógica del caso modelado y la información de su representación visual, y está escrito en Smalltalk con secciones de Alter embebidas dentro del código y que son interpretables en el entorno gráfico. En la Figura 7 se presenta una parte del código correspondiente al modelo de la Figura 6; allí se pueden diferenciar las siguientes partes:

1. Esta es la especificación del nodo clases. La interpretación del tipo de gráfico se encuentra en la especificación del meta-metamodelo; de esta manera el modelo sabe cómo se grafica, pero la posición correspondiente debe ser definida en esta parte del código.

2. Lo que se incluye en value es un método en lenguaje Alter para adicionar un elemento que se va a crear en el gráfico. El código se escribe entre apóstrofes porque sólo se interpreta en el entorno DOME.

3. Esta parte del código permite la creación de atributos. Nuevamente se incluyen elementos de la presentación, de la posición en la pantalla y de la identificación lógica de los diferentes elementos.

Como en el caso de AToM^3, no hay una separación clara entre la lógica del modelo y su representación de

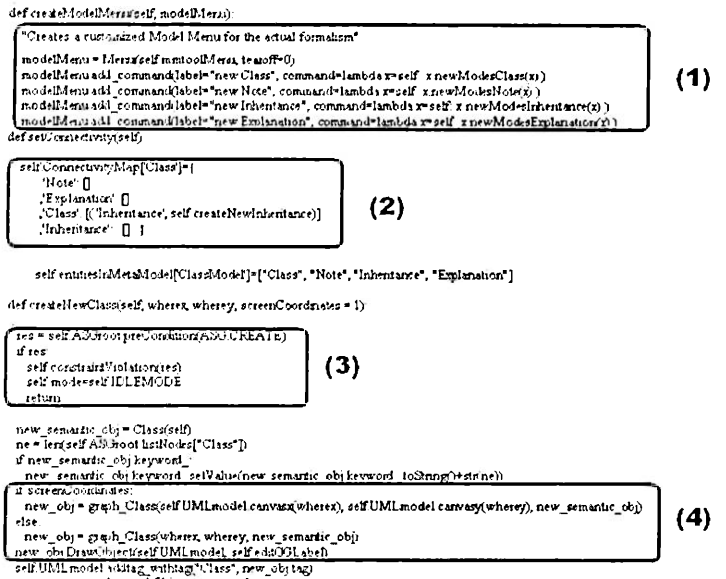


Figura 4: Parte del código python incluido en el archivo ClassModel_MM.py

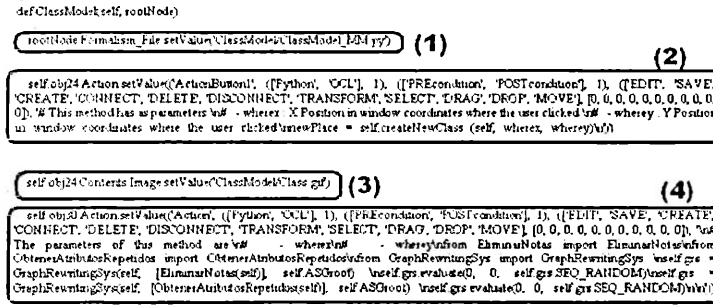


Figura 5: Parte del código python del archivo ClassModel.py

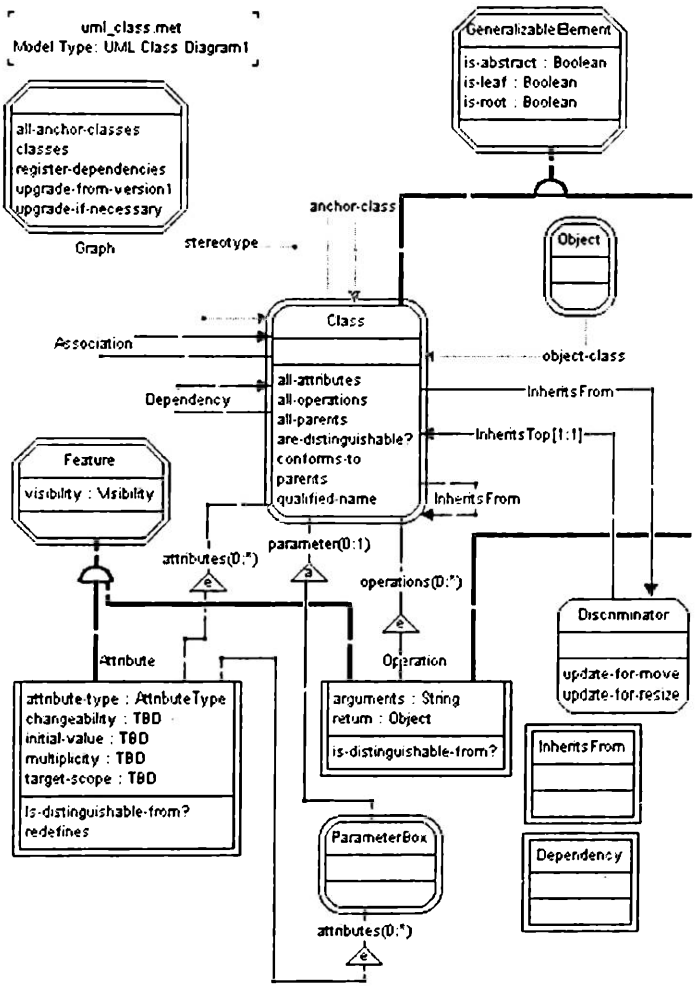


Figura 6: Modelo de clases expresado en el formalismo gráfico de DOME

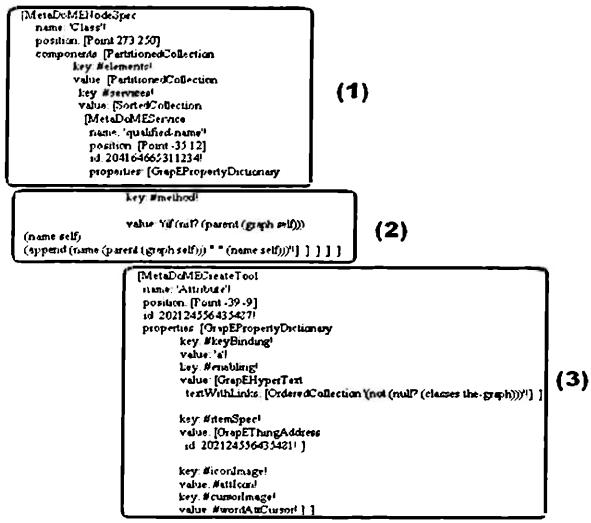


Figura 7: Parte del código Smalltalk que soporta la imagen de la Figura 6

tipo gráfico. Sin embargo, DOME abiertamente une ambas representaciones en un solo archivo, a diferencia de AToM^3, que tiene múltiples archivos y en todos ellos combina características gráficas con características lógicas del modelo.

Cuando se combina la lógica de representación de un metamodelo con su representación gráfica, se presentan problemas a la hora de cambiar la visualización del modelo; por ejemplo, si se sabe que las representaciones del diagrama de clases y del diagrama entidad-relación son muy similares, en un modelo independiente de la representación gráfica es directa la manera de conversión, puesto que sólo se eliminan los elementos del diagrama de clases que no están presentes en el modelo entidad-relación y se asigna la representación gráfica de cada elemento. En el caso de AToM^3 y DOME, esa conversión no se puede realizar directamente y requiere la programación de módulos especiales (en python en el primer caso y en smalltalk en el segundo) para realizar esa conversión entre diagramas.

En la siguiente sección se presenta UN-LEND como un formalismo que trata de independizar la lógica de representación de la representación visual de un metamodelo definido. De esta manera se entrega la flexibilidad al lenguaje de representación, permitiendo la utilización de diferentes representaciones gráficas para la misma representación lógica, uno de los aspectos en que fallan las herramientas metaCASE analizadas.

4 UN-LEND: LENGUAJE FORMAL PARA UN-METACASE

UN-MetaCASE es una herramienta que se encuentra actualmente en desarrollo por parte del grupo UN-INFO de la Escuela de Sistemas de la Universidad Nacional de Colombia. Para su construcción se han tomado en cuenta las deficiencias mostradas en la sección anterior, para definir como requisitos del lenguaje de representación los siguientes:

- Se requiere una separación entre la lógica de representación y la representación visual del metamodelo. UN-LEND debe atender la lógica de representación y ser independiente de las características gráficas del modelo.

- Se debe incluir un lenguaje para la expresión de restricciones, que le den expresividad al lenguaje y que permitan el cálculo de esas expresiones en el momento en que se generen instancias de los modelos.

4.1 Definición de UN-LEND

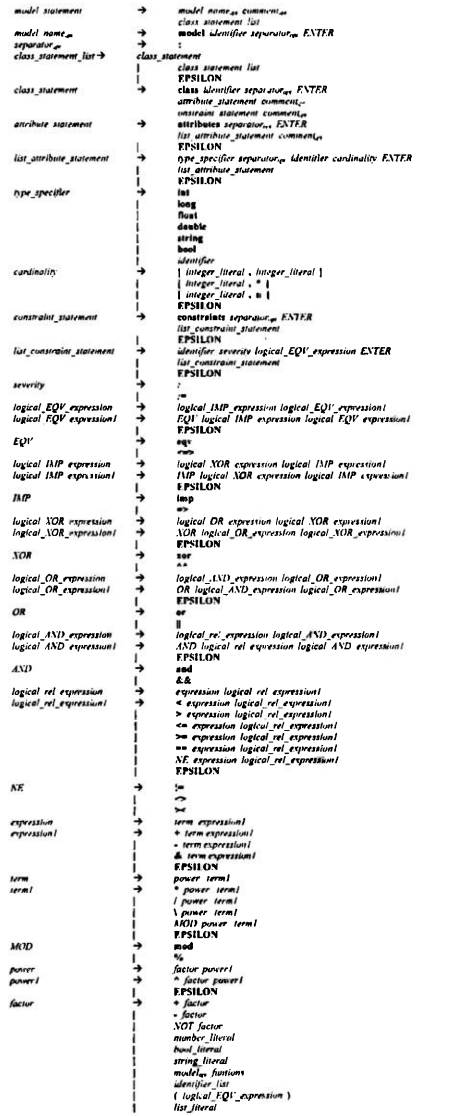
4.1.1 Elementos de la Estructura Léxica de UN-LEND

El lenguaje dispone de diferentes elementos, cuya sintaxis completa se puede consultar en [Alvarez (2001)]. Esos elementos incluyen:

- Alfabeto: las letras y símbolos comunes en cualquier alfabeto.
- Comentarios: Se incluyen con // y terminando con la tecla <ENTER> o con /* y */.
- Palabras reservadas: *and*, *attributes*, *bool*, *class*, *constraints*, *double*, *eqv*, *false*, *float*, *imp*, *int*, *list*, *Long*, *mod*, *model*, *Model*, *not*, *or*, *string*, *true*, *xor*
- Separadores: ( ) . , | | *space tab enter*
- Literales y Tipos de datos: numéricos (*int*, *long*, *float*, *double*), cadenas (string) y lógicos (booleanos, que pueden tomar el valor true/false).
- Operadores:
  - Aritméticos: +, -, *, /, (división entera), %, *mod* (los dos últimos para módulo de la división).
  - Relacionales: !=, <>, >< (los tres usados para distinto de), == (igual a), <, >, <=, >=.
  - Lógicos: negación (!, ~, *not*), conjunción (&&, *and*), disyunción (||, *or*), exclusión lógica (^^, *xor*), implicación (=>, *imp*), equivalencia (<=>, *eqv*).
  - Especiales: operador punto (.), el operador de concatenación (&), el operador de selección (||) y el operador de agrupación (()).
- Funciones predefinidas: valor absoluto (*abs*), tangente inversa (*atan*), raíz cúbica (*cbrt*), coseno (cos), conteo (*count*), disyunción lógica entre componentes (*exist*), exponencial (*exp*), conjunción lógica entre componentes (*forall*), dentro de una lista (*in*), conversión a minúsculas (*lcase*), longitud de cadena (*len*), logaritmo natural (ln), logaritmo (log), cadena sin espacios a la izquierda (*ltrim*), π (*pi*), número aleatorio (*rnd*), cadena sin espacios a la derecha (*rtrim*), selector de listas (*select*), validación de conjunto (*set*), seno (*sen*), raíz cuadrada (*sqrt*), conversión a cadena de caracteres (*str*), sumatoria (*sum*), tangente (tan), conversión de radiantes a grados (*toGrad*), conversión de grados a radianes (*toRadian*), concatenación de cadenas (*toString*), cadena sin espacios a la derecha y a la izquierda (*trim*), conversión a mayúsculas (*ucase*), conversión de cadenas a valores (*val*).

4.1.2 Sintaxis Formal de UN-LEND

La forma Backus-Naur (BNF) de la estructura de los lexemas correspondientes al lenguaje resultante se muestra en la Figura 8:



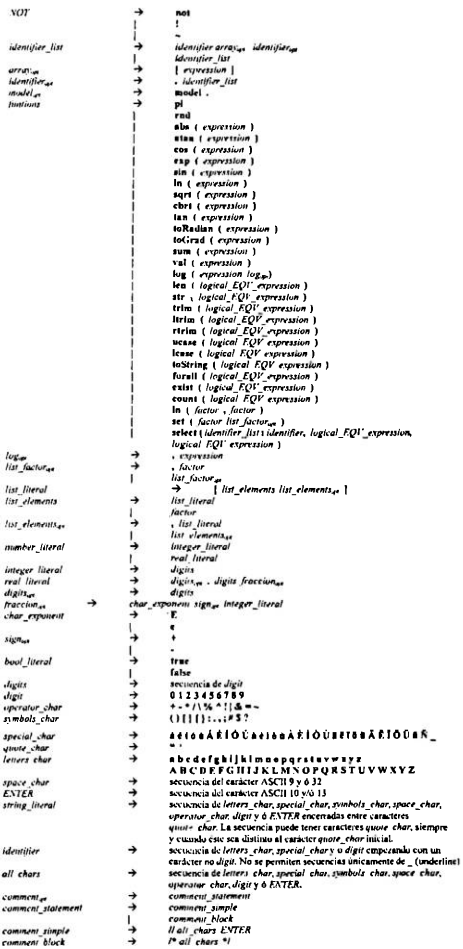


Figura 8: Forma Backus-Naur (BNF) del UN-LEND

Esta estructura se interpreta mediante un analizador sintáctico construido para que se pueda realizar la verificación de validez de las expresiones realizadas mediante UN-LEND y que además las tradujera a una estructura jerárquica para su posterior instanciación y verificación de restricciones.

4.2 Caso de Estudio

Para ejemplificar el manejo de UN-LEND, se ingresa en el parser el siguiente modelo correspondiente a la estructura de una familia:

```
class Hombre:
    attributes:
      string: nombre [1,2]
      string: apellido [1,2]
      edad [1,1]
    constraints:
      vive = edad > 0 and edad < 100
class Mujer:
    attributes:
      string: nombre [1,2]
      string: apellido [1,2]
      int: edad [1,4]
    constraints:
      vive = edad > 0 and edad < 100
class Casa:
    attributes:
      string: direccion [1,1]
      Hombre: dueno [1,1]
      string: telefono [1,*]
    constraints:
class Matrimonio:
    attributes:
      Hombre: esposo [1,1]
      Mujer: esposa [1,1]
      Casa: vivienda [1,*]
    constraints:
      MayorEsposo = esposo.edad > esposa.edad[2,0]
```

UN-MetaCASE posee un analizador sintáctico para los metamodelos expresado en UN-LEND y presenta en pantalla el informe que se muestra en la Figura 9. Posteriormente, se pueden ingresar las instancias de cada una de las clases presentes en el modelo, tal como se muestra en la Figura 10. Por último, se realiza la verificación de las restricciones correspondientes al modelo, empleando para ello la definición del modelo y el juego de instancias que se ingresaron, para finalmente obtener el reporte de errores de la Figura 11.

5 CONCLUSIONES

En el entorno de desarrollo del UN-MetaCASE se definió UN-LEND, un lenguaje formal para la expresión de modelos, que independiza la lógica de representación de los metamodelos de su representación visual y que tiene la capacidad de definir las restricciones del metamodelo incorporadas en el mismo lenguaje; estas características diferencian de manera fundamental al UN-MetaCASE de otras herramientas similares, como AToM^3 y DOME.

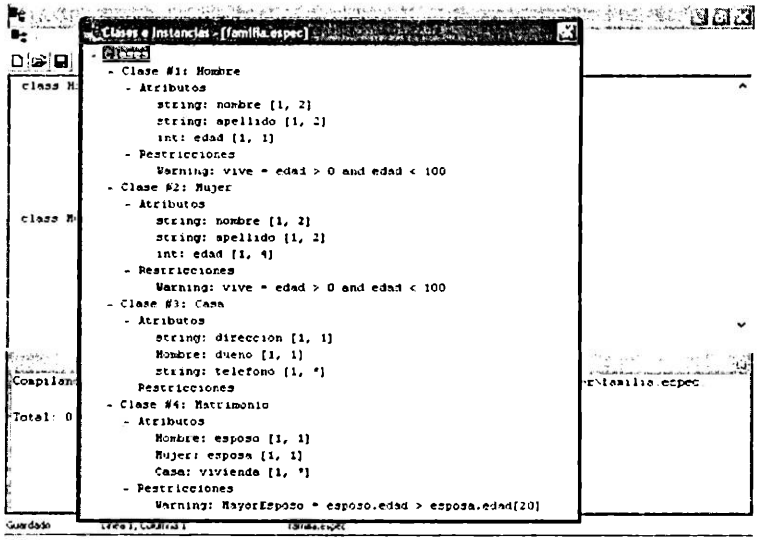


Figura 9: Imagen del modelo en la estructura jerárquica

Los metamodelos escritos en UN-LEND se interpretan mediante un analizador sintáctico que puede elaborar instancias del metamodelo y realizar la verificación de las restricciones correspondientes al mismo.

La independencia entre la lógica de representación y la representación visual contribuye a la evolución de los metamodelos, puesto que facilita la labor de conversión entre diferentes formas de representación sin que ello repercuta en cambios sobre la lógica de representación. Ello implica que, mediante el UN-LEND, los modelos escritos para el UN-MetaCASE tienen la facultad de utilizar la misma representación lógica para múltiples representaciones gráficas.

6 TRABAJO FUTURO

Se espera que la herramienta UN-MetaCASE continúe con su desarrollo, generando los mecanismos necesarios para ligar la lógica de representación con las características gráficas, pero sin recurrir para ello a la incorporación de dichas características en la especificación del UN-LEND.

En esta fase del trabajo, se deben elaborar los mecanismos necesarios para la comunicación entre el analizador sintáctico desarrollado para la interpretación del UN-LEND y el Editor Gráfico que permite la construcción de Modelos del UN-MetaCASE.

Es importante también continuar ampliando el lenguaje UN-LEND con otros elementos léxicos que faciliten la elaboración de metamodelos más complejos.

Se espera también iniciar la construcción del metamodelo de UML en UN-LEND, buscando con ello la traducción del formalismo en MOF y las reglas de consistencia y refinamiento de OCL en UN-LEND.

AGRADECIMIENTOS

Los autores desean expresar su gratitud a la Dirección de Investigaciones de la Sede Medellín de la Universidad Nacional de Colombia, pues este artículo se realizó dentro del marco del proyecto DIME No. 030805769, denominado "Construcción de un Editor Genérico para Elaboración de modelos gráfico-simbólicos - Etapa II - Establecimiento de la Consistencia entre los Tipos de Modelo generados mediante el MetaCASE".

REFERENCIAS

Alvarez, C. (2004), 'Módulo de UN-MetaCASE para la generación de instancias y la verificación de restricciones de una especificación orientada a objetos'. Trabajo Dirigido de Grado.

Bubenko, J., Langerfors, B. y Solvberg, A. (1971), Computer-Aided information systems analysis and design, Technical report, Lund.

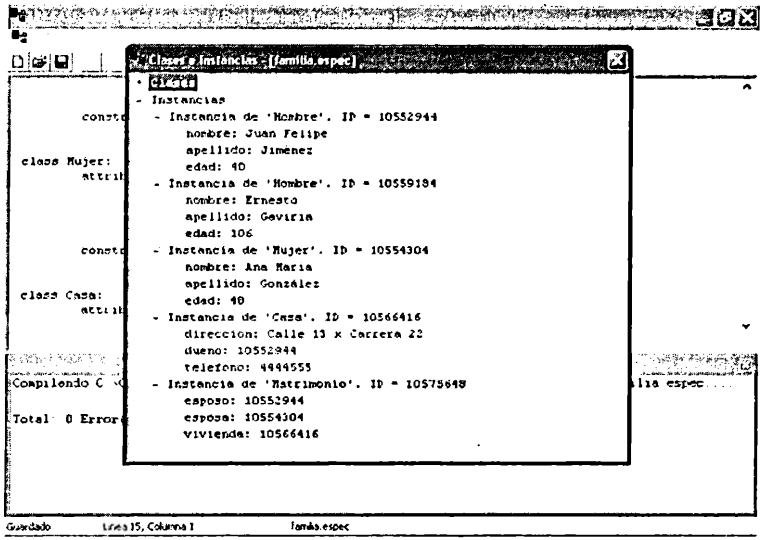


Figura 10: Imagen de las instancias generadas para el modelo de la Figura 9

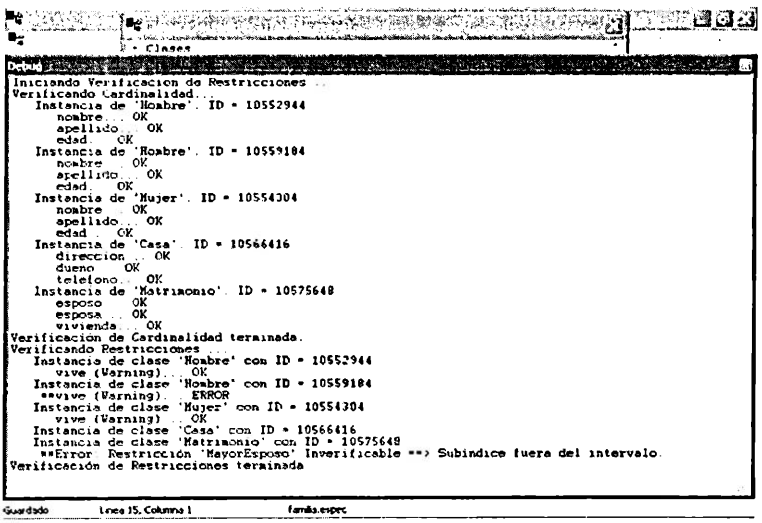


Figura 11: Imagen de la verificación de restricciones realizada con base en el modelo de la Figura 9 y las instancias de la Figura 10

Burkhard, D. y Jenster, P. (1989), 'Applications of computer-Aided software engineering tools: Survey of current and prospective users', *Data Base* **20**(3), 28–37.

De Lara, J. y Vangheluwe, H. (2002), AToM^3: A tool for multi-formalism and meta-modelling, *in* 'Proceedings of the Fifth Ingernational Conference on Fundamental Approaches to Software Engineering'. pp. 174-188.

DOME (2004), 'What is Dome'. En Línea: <http://www.htc.honeywell.com/dome/description.htm> C-10/04.

Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason IV, C., Nordstrom, G., Sprinkle, J. y Volgyesi, P. (2001), The generic modeling environment, *in* 'Proceedings of the Workshop on Intelligent Signal Processing, Budapest'.

OMG (2004), 'Object management group. UML 2.0 specification'. En Línea: <http://www.omg.org/uml> C-10/04.

PYTHON (2004), 'Phyton home page'. En Línea: <http://www.python.org> C-10/04.

SMALLTALK (2004), 'Smalltalk home page'. En Línea: <http://www.smalltalk.org> C-10/04.

Vangheluwe, H., De Lara, J. y Mosterman, P. (2002), An introduction to multi-paradigm modeling and simulation, *in* 'Proceedings of AI, Simulation and Planning in High Autonomy Systems (AIS), Lisbon'. pp. 9-20.

Carlos Mario Zapata es candidato a Doctor en Ingeniería de la Universidad Nacional de Colombia. Actualmente se desempeña como profesor Asistente en el área de Ingeniería de Software y ha conducido proyectos de investigación en Ingeniería de Software y Lingüística Computacional. Trabaja además en aplicación de técnicas de enseñanza - aprendizaje a las tecnologías de información.

Fernando Arango es Doctor en Ingeniería de la Universidad Politécnica de Valencia. Actualmente se desempeña como Profesor Asociado en el área de Ingeniería de Software y es Director de la Escuela de Sistemas de la Universidad Nacional de Colombia. Además, ha conducido proyectos de investigación en Ingeniería de Software, especialmente en métodos formales y conduce dos trabajos doctorales en el área de Ingeniería de Software.