XV Jornadas de Ingeniería del Software y Bases de Datos
JISBD 2006
José Riquelme - Pere Botella (Eds)

HACIA LA DEFINICION DE LENGUAJES ESPECIFICOS DE DOMINIO CON SINTAXIS GRAFICA Y TEXTUAL

Francisco Pérez, Juan de Lara

Departamento de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Ciudad Universitaria de Cantoblanco, 28049, Madrid
e-mail: {francisco.perez, jdelara}@uam.es

Palabras clave: Lenguajes Específicos de Dominio, Meta-Modelado, Gramáticas Libres de Contexto, Gramáticas de Grafos, Gramáticas de Pares, AToM^3.

Resumen. *En este artículo describimos nuestro enfoque para la asignación de sintaxis concreta tanto gráfica como textual a Lenguajes Específicos de Dominio (LEDs). Nuestra aproximación consiste en definir un meta-modelo que describe la sintaxis abstracta del LED, a cuyos elementos se puede asignar una visualización gráfica. Para asignarles una sintaxis textual se ha construido un meta-modelo con conceptos relevantes para la descripción de programas textuales (por ejemplo, operador, expresión, secuencia, etc). Se han definido una serie de transformaciones que, a partir del meta-modelo de la sintaxis abstracta del LED, generan un modelo conforme al meta-modelo para la sintaxis textual, que el diseñador del LED puede posteriormente refinar. A partir de este modelo, es posible generar un parser que crea instancias válidas del meta-modelo del LED a partir de programas textuales conformes a la gramática.*

1. INTRODUCCIÓN

Los Lenguajes Visuales Específicos de Dominio (LVEDs) son lenguajes de alto nivel, diseñados para ser eficientes en tareas particulares. Al estar restringidos a un dominio particular, ofrecen técnicas más potentes de análisis y generación de código, aumentando la productividad y mejorando la calidad de los productos generados [1].
Una de las técnicas más habituales para describir la sintaxis de los LEDs es mediante meta-modelado [2]. Un meta-modelo es un modelo de un lenguaje (de modelado) [3], que describe el conjunto de todos los modelos admisibles. Suele estar descrito en lenguajes tales como diagramas entidad relación o diagramas de clases, aunque frecuentemente es necesario añadir invariantes adicionales mediante lenguajes textuales de restricciones (como OCL).
Mientras que el meta-modelo y sus invariantes describe los elementos de la sintaxis abstracta de un LED, para su representación se hace necesaria la descripción de una sintaxis concreta

(es decir, cómo se ven los elementos de la sintaxis abstracta). En el caso de un lenguaje visual, en el caso más sencillo, basta con asignar iconos a las clases, y describir la apariencia de las asociaciones (ver por ejemplo [4]). No obstante, existen muchos lenguajes en los que resulta más natural expresarlos mediante una representación textual. Tal es el caso de OCL [5].

El modelado multi-formalismo [6] estudia la interconexión y transformación de modelos descritos en formalismos distintos, y a veces, en un mismo diagrama existe una mezcla de notaciones visuales y textuales, por lo que se hacen necesarios mecanismos para la manipulación homogénea de modelos textuales y visuales.

En este trabajo, presentamos un enfoque novedoso para la definición de una sintaxis concreta textual para LEDs. Para ello, hemos definido un meta-modelo *Textual* con los conceptos principales de los lenguajes textuales, tal es como *operador, secuencia,* etc. A partir del meta-modelo que describe la sintaxis abstracta de un LED generamos un modelo, instancia del meta-modelo *Textual*. A partir de este modelo *Textual* se genera la gramática EBNF que describe la sintaxis textual concreta, y con ésta implementamos un parser capaz de reconocer representaciones textuales de dicho LED. La traducción de esta representación a la representación abstracta del modelo es necesaria para su integración y procesamiento conjunto con otros modelos (posiblemente visuales) en sistemas multi-formalismo.

El enfoque propuesto se está integrando en la herramienta AToM^3, y permite el tratamiento unificado de lenguajes textuales y visuales. Hemos identificado varias transformaciones a *Textual,* que el diseñador puede elegir dependiendo del tipo de LED que se está tratando.

El resto del artículo está organizado de la siguiente manera. La sección 2 presenta nuestro enfoque para la generación de entornos para LVDEs, basado en la herramienta AToM^3. La sección 3 presenta el enfoque propuesto para la asignación de una sintaxis concreta textual a un LED, y presenta algunas transformaciones a *Textual*. La sección 4 muestra un ejemplo con la definición de una sintaxis textual para CTL. La sección 5 compara con trabajos relacionados, y finalmente la sección 6 termina con las conclusiones y el trabajo futuro.

2. DEFINICION DE LENGUAJES VISUALES ESPECIFICOS DE DOMINIO

En esta sección se describe nuestro enfoque para la generación automática de entornos para LVEDs con la herramienta AToM^3, y muestra la necesidad de contar con mecanismos que permitan también la definición de una sintaxis concreta textual.

El enfoque actual de AToM^3 se basa en utilizar meta-modelado para describir la sintaxis abstracta del LVDE, mientras que la sintaxis concreta se describe por medio de la asignación de una representación visual a cada clase y asociación del meta-modelo: iconos, y flechas respectivamente. Como ejemplo, la parte izquierda de la Figura 1 muestra la definición del meta-modelo para CTL en AToM^3, donde se está diseñando la representación visual de una de las clases por medio de un icono.

A menudo, el meta-modelo ha de completarse con restricciones para expresar ciertas condiciones que no pueden especificarse de manera visual. En AToM^3, esto se hace usando código Python que se evaluará en respuesta a eventos de usuario tales como crear, conectar, editar, etc. A partir de esta información, se genera una herramienta para el LVED definido.

Continuando con el ejemplo, a la derecha de la Figura 1, se muestra el entorno generado para CTL. Este entorno se genera automáticamente a partir del meta-modelo de la izquierda, y en él es posible definir, además de modelos acorde al meta-modelo, gramáticas de grafos para manipular los modelos. Esto evita al diseñador del LVED el conocer el API de la herramienta y el lenguaje de programación de ésta para poder expresar computaciones en los modelos.

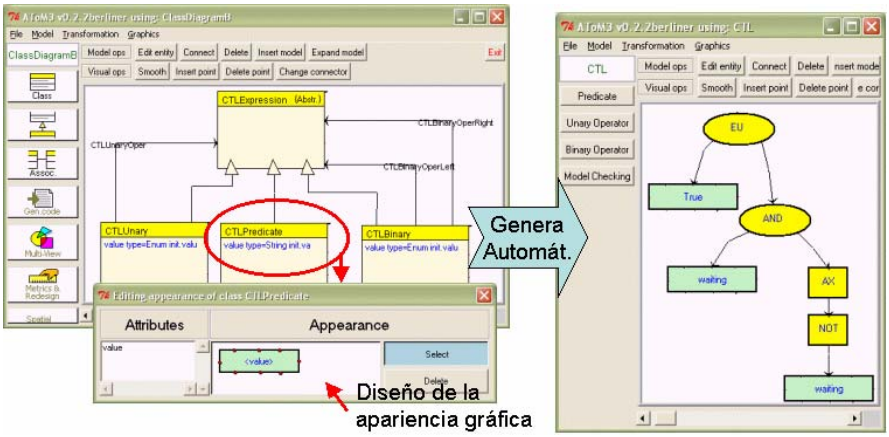

Figura 1. Definiendo el entorno para CTL (izquierda). Entorno generado (derecha)

Como puede verse en el ejemplo, a la derecha de la Figura 1, una sintaxis concreta visual no es la más adecuada para CTL, ya que el usuario tiene que dibujar el árbol sintáctico de la expresión deseada. En este caso una sintaxis concreta textual sería más adecuada. Se requiere por tanto una manera de asignar sintaxis textual a la sintaxis abstracta expresada por un meta-modelo, de tal forma que sea posible leer programas escritos en una sintaxis textual concreta, y producir a partir de ellos la representación abstracta de los modelos conforme al meta-modelo, de tal forma que sea posible integrados y procesados de manera homogénea con otros modelos, ya sean visuales o textuales. También se pretende que el diseñador del LED no tenga que ser experto en técnicas de compilación, o en gramáticas EBNF.

3. ENFOQUE PROPUESTO PARA LA DEFINICIÓN DE SINTAXIS TEXTUAL

Esta sección presenta un enfoque semi-automático para asignar sintaxis concreta textual a un meta-modelo. En primer lugar, se ha diseñado el meta-modelo de la Figura 2 con los conceptos principales de los programas textuales. Llamamos "*Textual*" al lenguaje descrito por este meta-modelo. Así, dado un meta-modelo que describe la sintaxis abstracta de un LED, generamos un modelo de *Textual* mediante la aplicación consecutiva de reglas de una gramática de grafos triple (TGG) [7].

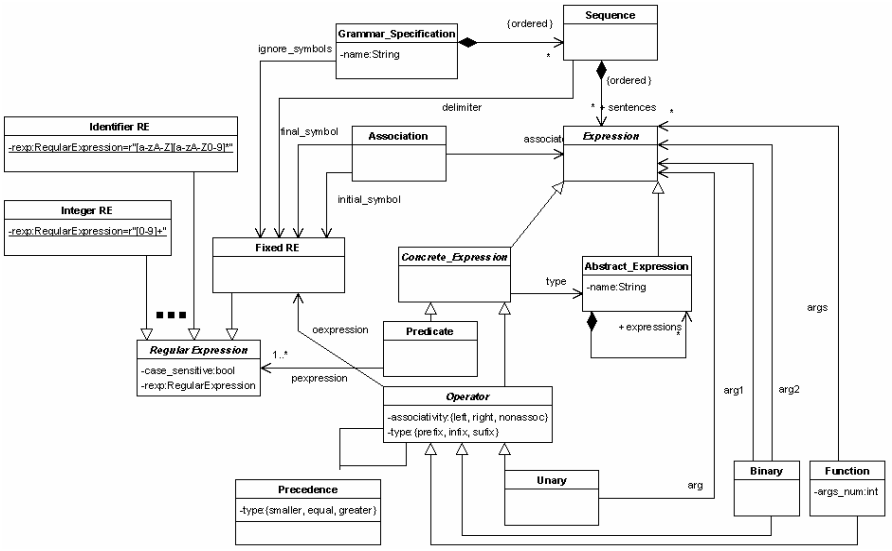

Figura 2. Meta-modelo para la definición de sintaxis concreta textual.

Un modelo del lenguaje *Textual* está formado por un único objeto de tipo *Grammar_Specification*, que contiene un nombre, una especificación de caracteres ignorables, y una secuencia ordenada de objetos de tipo *Sequence*. Estos objetos representan secuencias de expresiones, delimitadas entre ellas por *delimiter*. Las expresiones pueden ser concretas (*Concrete_Expression*) o abstractas (*Abstract_Expression*). Estas últimas permiten agrupar expresiones concretas bajo el mismo tipo. Las expresiones concretas pueden ser predicados u operadores, pero en cualquier caso siempre se representan mediante una expresión regular. Los expresión que representa los operadores debe ser constante (*Fixed_RE*), y en función del número de argumentos, pueden ser unarios, binarios, o funciones. Además, se puede definir su precedencia, su asociatividad (por la izquierda, por la derecha o no asociativo), y su posición respecto a los operandos (infijo, prefijo o sufijo), que son de tipo *Expression*. Los predicados se representan asimismo como una o más expresiones regulares de distintos tipos predefinidos, como por ejemplo identificadores (*Identifier RE*), números enteros (*Integer RE*), y otros que no mostramos por simplicidad.

La idea principal (que se muestra en la Figura 3) de nuestro enfoque es que, a partir de un modelo de *Textual*, y un mapping de esos elementos al meta-modelo del LED, es posible generar una gramática EBNF. Esta gramática se compila mediante la aplicación PLY [8], un compilador de compiladores que genera un *parser*, que a su vez se integra en AToM^3. Las acciones semánticas de la gramática generada construyen la representación abstracta del modelo conforme al meta-modelo original del LED. Estas acciones son en realidad reglas de gramáticas de grafos que se derivan durante la generación del modelo *Textual*. Esto es, seguimos el enfoque basado en el concepto de gramáticas de pares [9] propuesto por Pratt, que empareja reglas textuales y de grafos, y se utiliza para la transformación de texto a grafos. De esta manera, dada una representación textual de un modelo, el parser genera su sintaxis abstracta conforme al meta-modelo original. Estos pasos se muestran en la Figura 3.

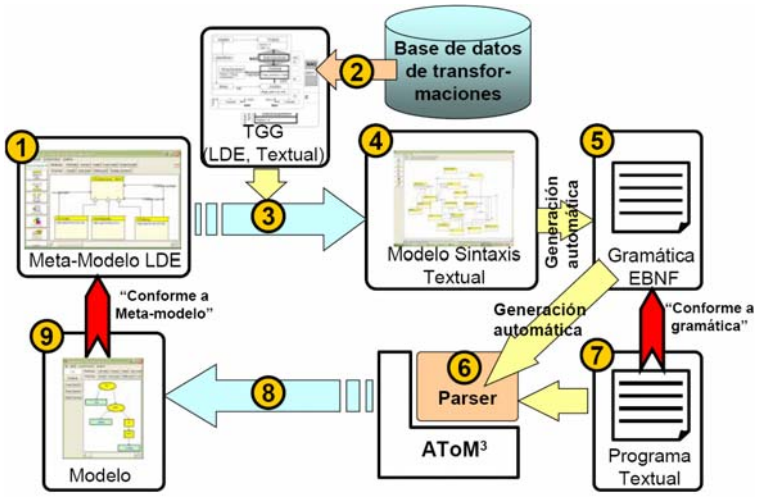

Figura 3. Proceso de generación y uso de la sintaxis concreta textual.

En el paso 1 de la Figura 3, el diseñador del LED especifica el meta-modelo para la sintaxis abstracta del lenguaje. Para asignarle una sintaxis textual, selecciona una transformación (paso 2) que produce un modelo *Textual* (etiquetado como 4), instancia del meta-modelo de la Figura 2. Estas transformaciones están descritas mediante TGGs y pueden variar dependiendo del tipo de LED con el que se esté trabajado. Por el momento estamos diseñado dos posibles transformaciones. Una de propósito general (aplicable a cualquier meta-modelo) que llamamos "*constructor/conector*". Esta crea una función por cada clase (que hace la vez de constructor de la clase), y asigna el resultado a una referencia. La función tiene un parámetro por cada atributo de la clase. Por otro lado, por cada asociación se crea otra función, con tantos parámetros como atributos tiene la asociación, más otros dos, correspondientes al origen y destino de la asociación. Dos de estas reglas se muestran en la Figura 4.

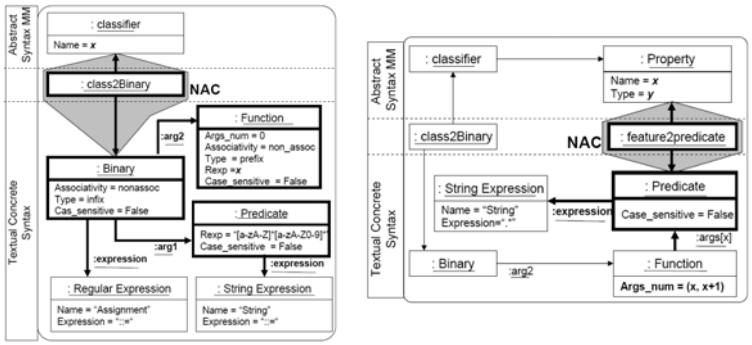

Figura 4. Dos de las reglas para los lenguajes tipo "constructor/conector"

Las reglas están divididas en tres partes que muestran, en la parte superior el meta-modelo del LED, en la inferior el modelo *Textual* que se está generando y en la mitad el grafo de

correspondencia (los mappings). Hemos utilizado una notación compacta para las reglas. Los elementos necesarios para que la regla se pueda aplicar se muestran en trazo fino. Los elementos que se crean como consecuencia de la aplicación de la regla se muestran en trazo grueso. Los elementos que no pueden aparecer para que la regla se aplique aparecen contenidos en polígonos sombreados y etiquetados como NAC (*negative application condition*). Más aún, si un atributo se modifica por la ejecución de una regla, éste aparece como una tupla, donde el primer valor es el que tenía el atributo antes de aplicar la regla, y el segundo es el valor que recibe al aplicar la regla.

La regla de la izquierda en la Figura 4 crea un operador binario por cada clasificador del meta-modelo (tanto clases como asociaciones). El operador binario es en realidad una asignación (objeto *Fixed RE*) que asigna a una referencia (objeto *Predicate*) el resultado de llamar al constructor (objeto *Function*), que se inicializa con cero parámetros. Una regla inicial que no mostramos por motivos de espacio, que previamente crea una instancia única de las clases que implementan las expresiones regulares para los tipos (*Identifier RE*, *Integer RE*, etc.)

La segunda regla crea un nuevo parámetro (objeto *Predicate*) por cada atributo no procesado del clasificador de tipo *Integer* e incrementa el número de parámetros de la función. Existen reglas similares por cada tipo distinto (*String*, *Float*, etc.).

La otra transformación es específica para LEDs con expresiones (aritméticas, lógicas, etc.) como por ejemplo CTL. Los meta-modelos de estos lenguajes contienen generalmente árboles de herencia, en los que las hojas son las clases significativas que se relacionan con clases padre (ver Figura 1). Las clases de estos lenguajes suelen tener un único atributo, que es su nombre o representación, y en función de las relaciones que tengan con otras clases se pueden considerar operadores u operandos. Las reglas de TGGs para estos lenguajes generan expresiones abstractas para las clases que no son hojas en el árbol, predicados para las hojas que no tienen relaciones con otras clases, y operadores unarios, binarios, o funciones según tengan una, dos o más relaciones con otras clases. Además, ya que este tipo de lenguajes suele asociar expresiones, se genera automáticamente una instancia de la clase *Association* relacionada con la expresión abstracta de la clase padre del meta-modelo. En la Figura 5 mostramos como ejemplo dos de las reglas (simplificadas para hacer más comprensible la presentación) de esta transformación. La de la izquierda asocia una expresión abstracta a cada clase. La de la derecha crea un objeto *Unary* asociado a una clase con una única asociación.

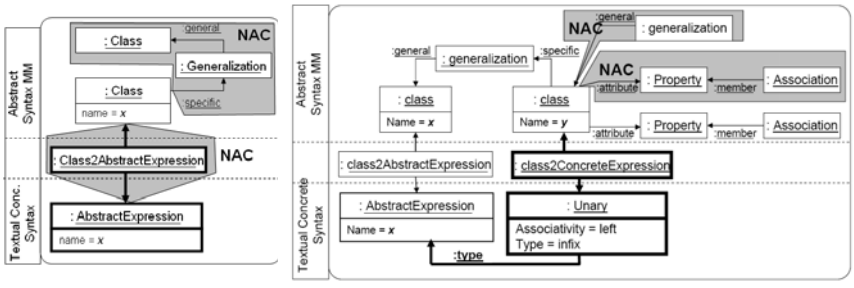

Figura 5. Dos de las reglas para los lenguajes tipo expresión.

La ejecución de estas transformaciones (paso 3 de la Figura 3) genera el modelo *Textual* etiquetado como 4. La transformación deja una serie de objetos en el grafo de correspondencia (*mappings*), que relacionan cada objeto del modelo *Textual* con el meta-modelo de la sintaxis abstracta. Nuestro objetivo es crear TGGs que generen sintaxis textuales concretas típicas, de tal manera que el proceso de crear una sintaxis concreta textual sea automático. No obstante, el modelo *Textual* resultante puede ser modificado y adaptado por el diseñador del LED.

El modelo *Textual* y los mappings permiten generar automáticamente una gramática EBNF (paso 5 en la Figura 3). Esta gramática contiene una serie de acciones semánticas formalizadas mediante reglas de una gramática de grafos. Así, cuando se emplea una regla EBNF, se ejecuta una regla de la gramática de grafos a la que la regla textual le pasa los parámetros correspondientes. Las reglas de la gramática de grafos se generan automáticamente a partir de los mappings entre el modelo *Textual* y el meta-modelo original. Esta aproximación se basa en las gramáticas de pares (pair grammars) [9].

Para la gramática EBNF hemos utilizado la sintaxis de PLY, un programa que genera un parser en Python (paso 6 en la Figura 3) para la gramática. El parser se integra en AToM^3, de tal manera que, cuando el usuario final introduce una representación textual conforme a la gramática EBNF, el parser (mediante la ejecución de las reglas de gramáticas de grafos que tienen asociadas las reglas textuales) genera la representación abstracta del modelo conforme al meta-modelo (paso 9 en la Figura 3).

Para un caso típico, el proceso de generar una sintaxis textual sólo implica que el diseñador del LED seleccione la transformación adecuada para el tipo de LED. El resto de pasos son automáticos y no requieren intervención del diseñador.

4. EJEMPLO

Continuando con el ejemplo de CTL (ver la izquierda en la Figura 1), observamos que cumple con las características de los lenguajes de tipo “expresión”, por lo que aplicaremos sobre él su conjunto de reglas triples. Como resultado se obtiene el modelo *Textual* de la Figura 6 (se han omitido algunos operadores unarios y binarios por simplicidad).

El modelo generado contiene un objeto expresión abstracta para la clase raíz del árbol jerárquico, y para cada clase hoja del meta-modelo original, en función del número de sus asociaciones, operadores unarios (una), binarios (dos), o predicado (cero). Se crea un objeto de cada tipo de operador para cada valor de la enumeración que tienen las clases, asociado a un objeto *Fixed_RE*. Las relaciones del meta-modelo de CTL se asocian al objeto de clase *Abstract_Expression* asociado a la clase padre. En el caso de la clase *Predicate* de CTL, se crea un objeto *Predicate* en *Textual*, cuya representación viene dada por una instancia de *Identifier_RE*. Finalmente, se crea una instancia de *Association*, para permitir agrupar subexpresiones, que utiliza como expresiones inicial y final de la asociación dos instancias de *Fixed_RE* que definen los paréntesis.

Una vez creado este modelo de forma automática, puede modificarse o refinarse para incluir por ejemplo la precedencia de operadores, que en este caso sería deseable para dar mayor prioridad a los operadores unarios que a los binarios. El siguiente paso es generar la gramática

EBNF asociada, con la cual PLY genera un parser. La figura 7 muestra parte de dicho código.

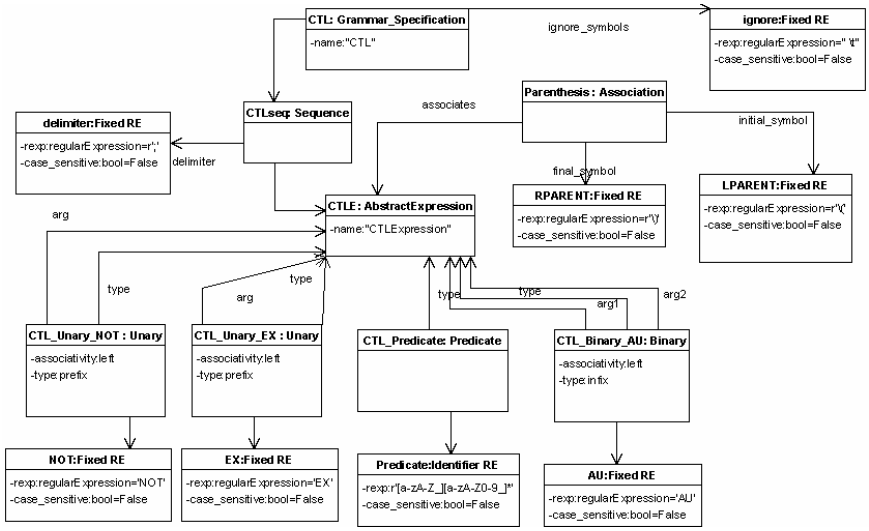

Figura 6. Modelo *Textual* de CTL (simplificado).

```
[1]  reserved_map = {}
[2]  tokens = ()
[3]  reserved_map['NOT'] = 'CTL_Unary_NOT'
[4]  tokens += ('CTL_Unary_NOT',)
     …
[5]  reserved_map['AND'] = 'CTL_Binary_AND'
[6]  tokens += ('CTL_Binary_AND',)
     …
[7]  tokens += ('CTLPredicate',)
[8]  def t_CTLPredicate(t):
[9]    r'[a-zA-Z_][a-zA-Z0-9_]*'
[10]   t.type = reserved_map.get(t.value, 'CTLPredicate')
[11]   return t
     …
[12] def p_CTLExpression(p):
[13]   '''CTLExpression : CTL_Unary_NOT CTLExpression'''
[14]   ret = CTL_Unary_NOT.execute (at3, at3.ASGroot, [(1, p[1]), (2, p[2])])
[15]   p[0] = p[1]
     …
```

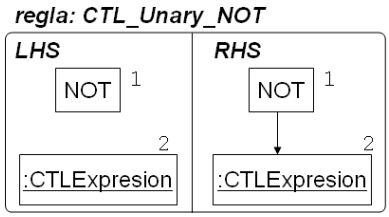

Figura 7. Extracto del código PLY generado.

Este código crea una serie de tokens, y muestra una de las reglas EBNF en la línea 12. Ésta contiene una llamada a una regla de gramáticas de grafos (*CTL_Unary_NOT*, que se muestra a la derecha) que conecta un operador *NOT* y un objeto *CTLExpression*. Esta regla recibe dos parámetros (p[1] y p[2], asociados a los objetos "1" y "2" de la regla de GG respectivamente) que son los dos elementos de la parte derecha de la regla EBNF, devolviendo el operador unario (línea 15). La regla de gramáticas de grafos *CTL_Unary_NOT* se genera automáticamente y usa un "objeto abstracto" (de tipo *CTLExpresion*). Naturalmente, no pueden existir instancias de clases abstractas en un modelo, pero un "objeto abstracto" en una regla puede identificarse con objetos de cualquiera de los subtipos de la clase abstracta [10].
Así finalmente, el modelo representado en la parte izquierda de la Figura 1, podría

representarse como: `"True EU waiting AND AX NOT waiting;".`

5. TRABAJO RELACIONADO

El presente trabajo mejora notablemente trabajo previo esbozado en [11]. En ese trabajo se proponía una transformación desde el meta-modelo del LED a un meta-modelo de más bajo nivel que el propuesto en la Figura 2. Además, en el presente artículo, proponemos varias transformaciones dependiendo del LED, usamos TGGs para la transformación, y las acciones semánticas son reglas de gramáticas de grafos que se pueden derivar automáticamente.

Muchos grupos de investigación han estudiado el problema de la conversión de programas a modelos y viceversa. Por ejemplo, en [12], justifican la imposibilidad de esta transformación entre meta-modelos y gramáticas debido a que no hay isomorfismo directo entre ambas, pero este sí es posible añadiendo información adicional. A pesar de ello, se da un algoritmo para las dos conversiones, pero no es adaptable al tipo de LED. En [13] buscan construir de forma semi-automática meta-modelos para gramáticas EBNF dadas, para lo que usan *anotaciones* con información adicional para permitir un isomorfismo. Otros enfoques, como [14] se basan en la UML Human-Usable Textual Notation (HUTN) de OMG [15]. HUTN se diseñó para proveer de una sintaxis textual "amigable" a meta-modelos, que se puede configurar hasta cierto punto mediante patrones, pero todos ellos fuera de la información contenida dentro de los modelo. Por ello, consideramos que nuestro enfoque es mucho más flexible (permite una adaptación mucho mayor a tipos concretos de LED), mediante la definición explícita de una librería de TGGs que generan modelos de *Textual*, y autocontenida, ya que usamos meta-modelado para la información adicional de la representación textual necesaria.

Nuestro enfoque es original ya que: a) Provee una biblioteca de transformaciones para diferentes tipos de LEDs. b) Permite al diseñador elegir la transformación más adecuada. c) Expresa las transformaciones a la sintaxis textual mediante TGGs, lo que facilita su comprensión y mantenimiento, d) Las acciones semánticas de las reglas EBNF son reglas de gramáticas de grafos, lo que también las hace fácilmente entendibles y mantenibles.

6. CONCLUSIONES Y TRABAJO FUTURO

En este artículo hemos presentado un enfoque novedoso para la asignación de sintaxis concreta textual a LEDs especificados mediante meta-modelado, gracias a la generación de parsers específicos para cada lenguaje. El enfoque es formal, ya que se basa en gramáticas de grafos, gramáticas triples y gramáticas EBNF. La idea es hacer accesible al diseñador del LED varias transformaciones a sintaxis textuales, y que éste elija la que mejor se adapte a su LED. Por supuesto, el diseñador puede describir sus propias transformaciones.

El trabajo futuro se centra en facilitar la integración del parser en AToM^3. De momento, nuestro enfoque funciona en una dirección: a partir de una representación textual se genera la representación abstracta del modelo. Sería interesante hacer posible que, cuando el usuario final modifique el modelo, se modificara el programa textual. Esto debería ser posible ya que se han emparejado reglas EBNF y de gramáticas de grafos. De esta manera, se podrían utilizar las reglas de gramáticas de grafos para realizar un parsing del modelo, que ejecutara las reglas EBNF asociadas.

Se pretende integrar este enfoque con las capacidades de generación de entornos multi-vista de AToM^3. Así, se podrán definir lenguajes donde algunas de sus vistas sean textuales, otras gráficas e incluso otras que combinen ambas representaciones en la misma vista.

Agradecimientos: Este trabajo ha sido subvencionado en parte por el Ministerio de Educación y Ciencia, proyecto MOSAIC (TSI2005-08225-C07-06), y por la Consejería de Educación de la Comunidad de Madrid y el Fondo Social Europeo (F.S.E.), mediante una beca FPI. Los autores quieren agradecer además los comentarios y sugerencias de los revisores.

REFERENCIAS

[1] Luoma, J., Kelly, S., Tolvanen, J.-P. "*Defining Domain-Specific Modeling Languages: Collected Experience"*. OOPSLA Workshop on DSLs (2004).

[2] Atkinson, C., Kühne, T. "*Rearchitecting the UML infrastructure"*. ACM Transact. on Modeling and Computer Simulation, Vol. **12**(4), pp.: 290-321, (2002).

[3] Favre, J. M. "*Towards a Basic Theory to Model Model Driven Engineering"*. Workshop on Software Model Engineering (WISME) en UML2004 Lisboa (2004).

[4] de Lara, J., Vangheluwe, H. "*AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling"*. FASE'02, Springer LNCS 2306, pp. 174 – 188 (2002).

[5] Warmer, J., Kleppe, A. "*The Object Constraint Language: Getting Your Models Ready for MDA, 2nd Edition*". Pearson Education. Boston, MA (2003).

[6] Vangheluwe, H., de Lara, J., Mosterman, P. J. "*An Introduction to Multi-Paradigm Modelling and Simulation"*, AI Simulation and Planning 2002. pp.: 9-20. Lisbon.

[7] Schürr, A. "*Specification of Graph Translators with Triple Graph Grammars"*. In LNCS 903, pp.: 151-163. Springer (1994).

[8] Página web de PLY Lex-yacc: http://www.dabeaz.com/ply/

[9] Pratt, T. W. "*Pair grammars, graph languages, and string-to-graph translations"*. Journal of Computer and System Sciences **5** pp.: 560-595 (1971).

[10] de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., y Taentzer, G. "*Attributed Graph Transformation with Node Type Inheritance*". Aceptado en Theoretical Computer Science (Elsevier).

[11] de Lara J., Guerra E. "*Towards the Uniform Manipulation of Visual and Textual Languages in AToM^3"*. Actas de PROLE'2003, pp.: 45-58. Alicante (2003).

[12] Alanen, M., Porres, I. "*A Relation Between Context-Free Grammars and Meta Object Facility Metamodels*". Informe Técnico 606, TUCS - Turku Centre for Computer Science, Turku, Finland, Mar (2004).

[13] Wimmer, M., Kramler, G. "Bridging Grammarware and Modelware". WiSME 2005, 4th Workshop in Software Model Engineering (2005)

[14] Muller, P.-A., Hassenforder, M. "*HUTN as a Bridge between Modelware and Grammarware – An Experience Report*". WiSME en MoDELS'05, Jamaica (2005).

[15] Página web de HUTN: http://www.omg.org/cgi-bin/doc?formal/2004-08-01