Arquitectura y Modelado de un Compilador de Lenguaje para Programas Disyuntivos

Juan J. Gil y Aldo Vecchietti
Universidad Tecnológica Nacional – Regional Santa Fe
INGAR – Instituto de Desarrollo y Diseño
Avellaneda 3657 – 3000 Santa Fe – Argentina
e-mail: jgil@frsf.utn.edu.ar – aldovec@ceride.gov.ar

Resumen. En este trabajo se describen los aspectos más destacados y las experiencias extraídas en el modelado y desarrollo de un compilador de un lenguaje para la formulación de programas disyuntivos. Los programas disyuntivos son una alternativa a los programas de optimización matemáticos que involucran restricciones con decisiones discretas. Además del compilador del lenguaje se han implementado los algoritmos necesarios para resolver los problemas mencionados. También se presenta la conexión y comunicación entre el compilador y los programas de solución de los problemas disyuntivos. El lenguaje propuesto se implementa como un superconjunto del lenguaje del sistema de resolución de programas matemáticos GAMS. Las interacciones entre ambos sistemas se detallan en este trabajo.

1. Introducción

En los últimos años se han realizado grandes esfuerzos para la inclusión de lógica en los programas de optimización matemáticos. La Programación con Restricciones Lógicas (PRL) (en inglés conocida como Constrained Logic Programming – CLP) y la Programación Disyuntiva Generalizada (PDG) (en inglés Generalized Disjunctive Programming – GDP) son ejemplos de áreas de investigación relacionados con la inclusión de lógica en la formulación de programas matemáticos. El empleo de lógica en la formulación de los problemas matemáticos facilita en muchos casos la expresión, lectura y entendimiento de restricciones con decisiones discretas. Muchas decisiones discretas se escriben de un modo más natural por medio de restricciones lógicas y/o disyunciones.

Por otra parte, los algoritmos de solución propuestos por PRL y PDG han mejorado en muchos casos el tiempo de ejecución y la solución óptima alcanzada comparada con aquellos algoritmos propios de los programas matemáticos. (Darby-Dorman y otros., 1997, Van Hentenryck y Saraswath, 1997, Turkay y Grossman, 1996, Lee y Grossmann, 2000). Además, con estos nuevos algoritmos se pueden abordar problemas más grandes y complejos. Aún con las ventajas que se pueden obtener con PRL y/o PDG se conocen pocas implementaciones comerciales o de investigación de sus algoritmos. ECLIPSE (Wallace y otros., 1997) e ILOG (ILOG, 1998) son sistemas implementados para resolver problemas del área de PRL, LOGMIP (Vecchietti y Grossmann, 1999) es un prototipo de investigación implementado para problemas del área PDG. Una de las principales dificultades para la implementación de tales sistemas es que no existe un lenguaje para la formulación de disyunciones y restricciones lógicas. Los sistemas para la resolución de programas matemáticos, por ejemplo GAMS, AMPL, LINDO, que son los más conocidos y usados no están preparados para tales formulaciones. Por lo tanto es necesario implementar un lenguaje que permita la incorporación de disyunciones y proposiciones lógicas en los programas matemáticos (Vecchietti y Grossmann 2000). La inclusión en un sistema de resolución de programas matemáticos surge porque PRL y PDG completan a la programación matemática pero no la sustituyen.

En este trabajo presentamos los aspectos relevantes relacionados con la implementación del lenguaje para la expresión de disyunciones en programas de optimización disyuntivos. En primer lugar se presenta la arquitectura del compilador y se describen los modelos de su representación interna. Luego se detallan las soluciones adoptadas para resolver la interacción entre el sistema existente (GAMS) y el nuevo (LOGMIP).

2. Programas Disyuntivos

La forma más básica y común de formular un problema de optimización matemático con decisiones discretas es aquel que se representa en forma algebraica de la siguiente manera:

$$\begin{aligned} & \min Z = f(x, y) \\ s.a. \quad & g_j(x,y) \leq 0 \quad j \in J \\ & x \in X, \; y \in Y \end{aligned} \qquad \textbf{(MILP ó MINLP)}$$

donde $f(x,y)$, $g(x,y)$ son funciones diferenciables convexas, J es el conjunto de índices de las desigualdades, x e y son variables continuas y discretas, respectivamente. El hecho que la formulación anterior corresponda a un programa Mixto-Entero Lineal (MILP) o Mixto–Entero No Lineal (MINLP) se debe a si las funciones $f(x,y)$, $g(x,y)$ son lineales o no lineales, respectivamente. X comúnmente se considera como un conjunto compacto convexo:

$$X = \left\{ x \,/\, x \in R^n, Dx \leq d, x^{lo} \leq x \leq x^{up} \right\}$$

donde x^{lo} y x^{up} son la cota inferior y superior de la variable x respectivamente, mientras que Y corresponde a un conjunto discreto poliédrico de puntos enteros:

$$Y = \left\{ y \,/\, y \in Z^m, Ay \leq a \right\}$$

que en la mayoría de las aplicaciones está restringido a valores de 0-1, $y \in \{0,1\}^m$.

En particular, el programa mixto entero (MINLP) también se puede formular como un programa disyuntivo generalizado, de acuerdo a lo propuesto por Raman y Grossmann (1994):

$$\begin{aligned} \min \quad & Z = \gamma_k c_k + f(x) \\ \text{sujeto a:} \quad & \\ & g(x) \leq 0 \\ & \bigvee_{i \in D_k} \begin{bmatrix} Y_{ik} \\ h_{ik}(x) \leq 0 \\ c_k = \gamma_{ik} \end{bmatrix} \quad k \in SD \qquad \textbf{(PDG)} \\ & \Omega(Y) = True \\ & x \in R^n \; , \; Y_{ik} \in \{True, False\}^m, \; c_k \geq 0 \end{aligned}$$

en donde Y_{ik} son las variables booleanas que establecen si un término dado en una disyunción es verdadero [$h_{ik}(x) \leq 0$], mientras que $\Omega(Y)$ son las relaciones lógicas expresadas en forma de lógica proposicional, que incluyen sólo variables booleanas. Y_{ik} son variables auxiliares que controlan la parte del espacio factible en la que se encuentran las variables continuas, x, y las variables c_k representan costos fijos que se activan a un valor γ_{ik} si el término correspondiente de la disyunción

es verdadero. Finalmente, las condiciones lógicas, *Ω(Y)*, expresan relaciones entre los conjuntos disyuntivos y por lo general pueden expresarse en Forma Conjuntiva Normal (FCN):

$$\underset{l=1,2,...L}{\wedge}\left[\underset{(j,k)\in P_l}{\vee}(Y_{ik})\underset{(j,k)\in Q_l}{\vee}(\neg Y_{ik})\right]$$

donde P_l es el subconjunto de variables Booleanas Y_{ik} que son verdaderas, y Q_l es el subconjunto de variables Booleanas que son falsas en las cláusulas l, l=1,2,…,L. En el problema *(PDG)* $f(x)$, $g(x)$ y $h_{ik}(x)$ se asumen convexas y diferenciables, también se asume que el problema *(PDG)* tiene una región factible compacta y que cada término de la disyunción tiene una región factible no-vacía.

El lector puede encontrar en Raman y Grossmann (1994) una descripción completa acerca de las disyunciones, sus propiedades y las implicancias que tienen en un programa matemático.

3. Lenguaje para la expresión de disyunciones

Los siguientes criterios se tuvieron en cuenta para la selección de las sentencias que se emplearon para formular disyunciones:

- Sintaxis simple
- Semántica muy ligada a la expresión de disyunciones, de modo que se puedan modelar bloques que sean exclusivos entre ellos.
- La sintaxis debe ser conocida para un usuario común.
- La sentencia debe permitir la definición de disyunciones embebidas (disyunciones que contengan disyunciones).

Se eligió la sentencia IF..THEN..ELSE..ENDIF porque satisfacía cada una de las características previamente mencionada, y además no estaba implementada en el lenguaje GAMS para otros usos.

La siguiente notación se emplea para describir la sintaxis del lenguaje:

Símbolo	Significado
< >	La frase encerrada corresponde a una regla sintáctica
Palabra	Símbolo terminal
[]	Expresión Opcional
{}	Expresión que puede repetirse
( )	Expresión que puede ser agrupada
::=	Definido como
|	O

Para la expresión de disyunciones en LOGMIP se emplea la siguiente gramática libre de contexto:

<Modelo LOGMIP> ::=
(<Declaración Disyunción> | <Definición Disyunción >) {**;** <Modelo LOGMIP>}
;
<Declaración Disyunción>::=
Disyunción identificador {**, identificador**} **;**
;
< Definición Disyunción > ::=
Identificador Disyunción is <Sentencia If>
;
<Sentencia If> ::=
if <condición> **then** **<**componentes> **else** <componentes> **end if ;**
| **if** <condición> **then** **<**componentes> {**elsif** **<**condición> **then** **<**componentes>} **end if** ;
<componentes> ::=
entidad {**; entidad**} [**;** <Sentencia If>];

En la sintaxis presentada <componentes> representa los identificadores de las restricciones del lenguaje matemático básico (LMB). Estos identificadores pueden ser representados por los LMB en diferentes formatos. Para poder aislar la sintaxis del lenguaje LOGMIP de las diferentes sintaxis que los LMB pueden adoptar, se definió la regla de los componentes como una secuencia de entidades. El propósito final de esto es que LOGMIP pueda ser incluido en cualquier LMB, no solo GAMS sino también LINDO, AMPL, etc. y que esta inclusión tenga un mínimo impacto en la gramática LOGMIP.

4. Arquitectura del Compilador

El compilador se diseño empleando la arquitectura de Pipes&Filters (Garlan y Shaw, 1993). Son dos los principales componentes de esta arquitectura: a) los *"filters"* que realizan alguna transformación sobre la información (entrada) que ellos reciben y producen como resultado (salida) alguna otra información y b) los *"pipes"* que tienen como función el transporte de información entre los *"filters"*. La arquitectura del compilador LOGMIP basada en este modelo se puede ver en la Fig. 1.

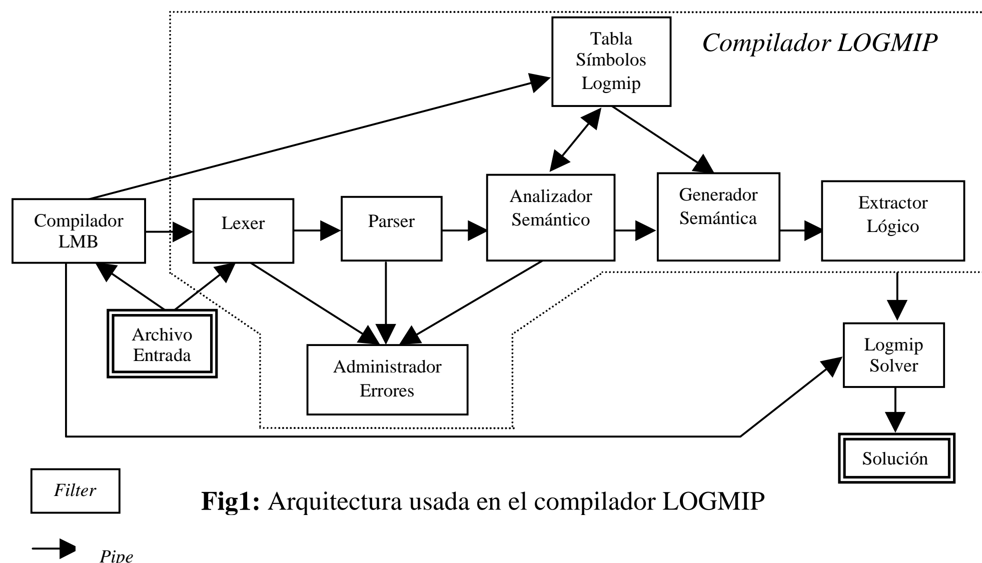

Fig1: Arquitectura usada en el compilador LOGMIP

Las tabla siguiente muestra los diferentes *"filter"* que componen el compilador y la función que tienen:

Filters	Función
Lexer	Transforma la entrada en símbolos terminales que se pasan al parser.
Parser	Verifica que la secuencia de símbolos terminales sea correcta y genera un modelo intermedio que se pasa al analizador semántico.
Analizador Semántico	Verifica que el modelo semántico intermedio sea correcto. En este punto toda la validación acerca del modelo de disyunciones termina.
Generador Semántico	Transforma el modelo validado en una representación de disyunciones LOGMIP y la pasa al Extractor Lógico.
Extractor Lógico	Toma la representación de las disyunciones LOGMIP y genera un archivo con toda la información lógica necesaria para el Solver LOGMIP.
Tabla Símbolos LOGMIP	Recibe la información de la tabla de símbolos del LMB y las transforma en identificadores LOGMIP
Administrador de errores	Recibe la información de los errores léxicos, sintácticos y semánticos encontrados y los muestra al usuario

5. Diseño de los componentes de la arquitectura

El análisis semántico, solo es realizado si el análisis léxico y sintáctico de la entrada no encontró errores en la misma, en caso contrario se considera que el modelo de entrada es inválido y, por lo tanto, no es necesario verificar si el significado de las proposiciones que lo conforman es correcto.

De aquí surge la necesidad de asociar dos posibles estados de reconocimiento del parser, representados en la Fig 2, a saber:

a) ReconocimientoModeloLogmip (estado inicial): implica que hasta el momento, el parser no ha encontrado un error en la conformación de la entrada en el análisis léxico/sintáctico realizado. A medida que el parser avanza en este estado, se va generando la representación interna de la entrada analizada (una estructura basada en objetos). Si el parser termina el análisis y se halla en este estado, entonces la entrada es correcta y se debe analizar semánticamente el modelo interno producido.

b) BúsquedaErrores: solo se puede llegar a este estado luego de haber reconocido un error léxico o sintáctico. En el momento que se produce este evento, el modelo interno pasa a ser inválido y, por lo tanto, se destruye para evitar posibles inconsistencias. El parser debe dejar de generar un modelo interno y solo tratar de obtener el resto de errores sintácticos o léxicos que puedan quedar en la entrada. Si el parser termina su fase en este estado, se advierte al usuario de los errores encontrados y se aborta la compilación LOGMIP.

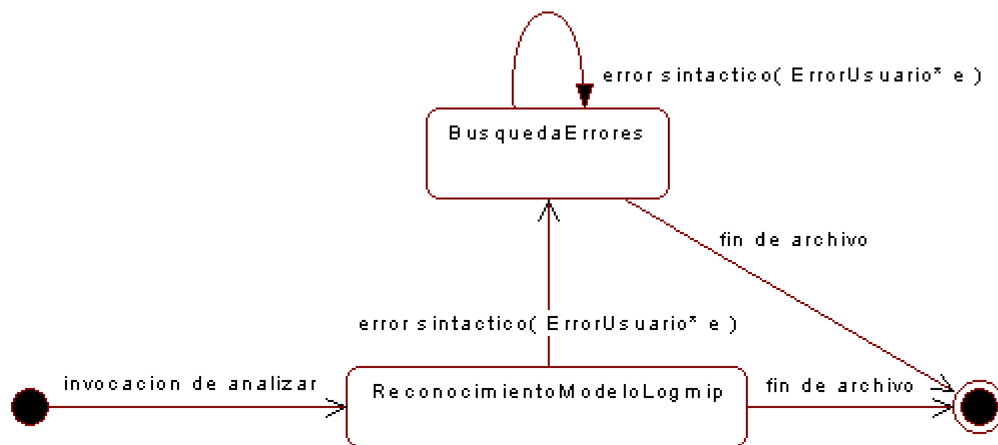

Fig. 2: Diagrama de estados correspondiente al analizador sintáctico

El comportamiento del analizador, como se vio anteriormente, depende de su estado actual. Esta posible variación de comportamiento, se modeló mediante una implementación del patrón de diseño *state* la cual permite variar el comportamiento de un objeto en tiempo de ejecución.

A los estados factibles del parser se les responsabilizó por la creación del modelo interno y de las posibles transiciones entre estados. De esta manera, las acciones a realizar una vez reconocida una construcción delegan al estado el *que* hacer con la misma y como tratarla.

Además, los estados de parsers, al mantener solo posibles comportamientos (algoritmos asociados al estado de análisis) y no información (atributos) fueron implementados como *singletons* (patrón de diseño Singleton o Instancia Única), donde se asegura que una clase posee, a lo sumo, una sola instancia en todo el sistema. Esta representación se puede ver en la Fig. 3.

En la Fig. 3 se puede ver como el estado de reconocimiento del modelo LOGMIP se ve representado por la clase *EstadoReconocimientoModelo* y el estado de *BúsquedaEerrores* se ve representado por *EstadoBusquedaErrores*. La forma que se comporta el parser se ve mantenida por estos estados y se logra una gran independencia de la herramienta utilizada para generarlo.

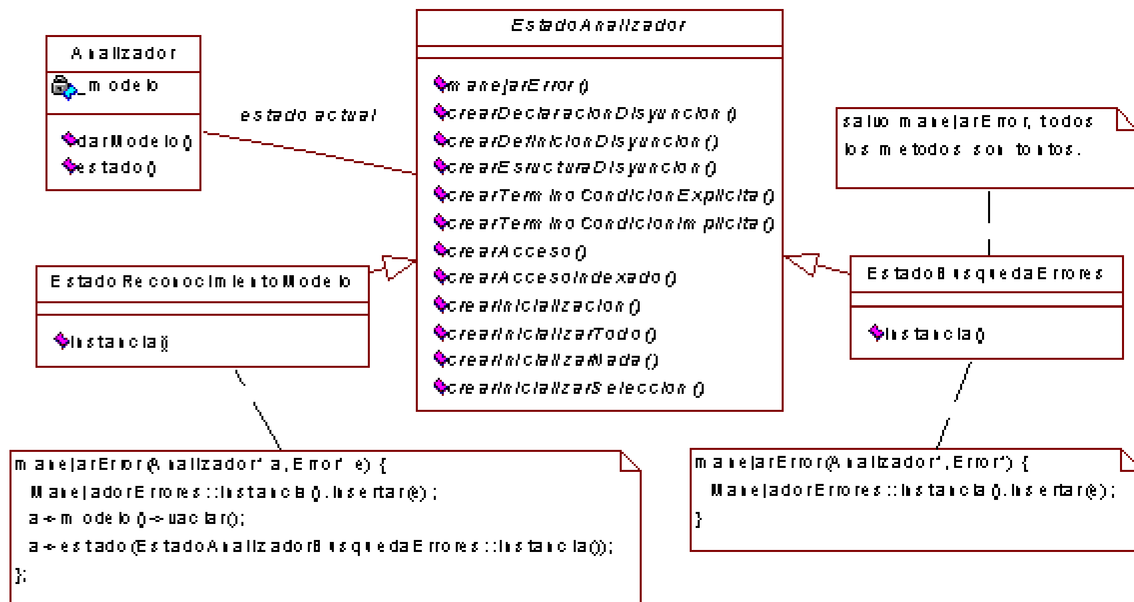

Fig. 3: Implementación del patrón Estados para el Analizador Sintáctico

5.1. El modelo de representación interna inicial (Modelo Sintáctico LOGMIP).

Como ya se dijo, una vez terminado el análisis sintáctico se tiene una estructura de objetos que representa el modelo descripto en la entrada del analizador sintáctico. Esta estructura, es una representación de las proposiciones LOGMIP encontradas en el modelo. Esto significa que cuando el parser reconoce una construcción sintáctica, genera su correspondiente representación en objetos y la agrega a esta estructura.

Las abstracciones fueron modeladas a partir de la descripción de una gramática libre de contexto donde se habla de *Símbolos* que pueden ser *terminales* (usualmente denominados tokens, o símbolos indivisibles) o *no terminales* (elementos de la gramática que se definen a partir de otros símbolos que pertenecen a ésta). De esta manera se genera una estructura de árbol de símbolos gramaticales *terminales* y *no terminales*. Para describir esta situación, se utilizo el patrón de diseño "Composite" (Compuesto) el cual permite describir este tipo de estructuras arbóreas donde todo elemento puede formar parte de otro y se necesita tratarlos a todos de la misma manera. Este esquema se puede ver en la siguiente Fig. 4:

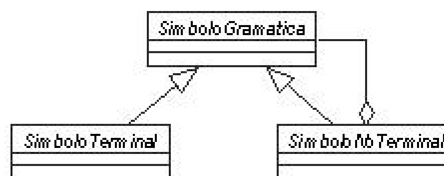

Fig. 4: Clases Bases para la representación Sintáctica de LOGMIP

Las abstracciones del modelo anterior se extendió para lograr una representación mas eficiente de los posibles símbolos que componen un modelo LOGMIP. El resultado de esta extensión se puede ver en la Fig 5.

A partir de estas abstracciones básicas, se implementaron las clases finales que representan los símbolos necesarios para definir la gramática LOGMIP (palabras reservadas, operadores, declaraciones y definiciones de disyunciones, inicializaciones, condiciones lógicas, lógica proposicional, etc.).

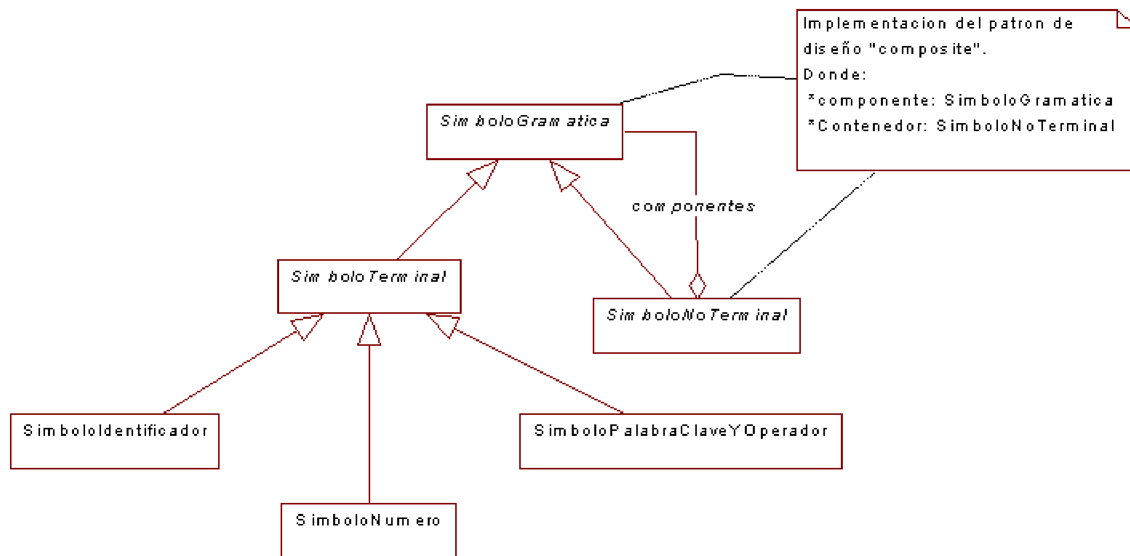

Fig 5: Implementación de COMPOSITE para la representación Sintaxis LOGMIP

En toda gramática libre de contexto existe un símbolo no terminal especial denominado simbolo de comienzo, mediante el cual se determina si una secuencia de símbolos pertenece o no al lenguaje descripto por la gramática. Este símbolo es representado por la clase RModeloLM (por Regla de Modelo LOGMIP) y es la abstracción que contiene la secuencia de proposiciones que representan al modelo. Entonces, si el parser determina que la entrada forma parte del lenguaje LOGMIP, mantiene la representación del mismo en una instancia de la clase RModeloLM.

5.2. Verificación Semántica

Esta fase del compilador se realiza solo si la verificación sintáctica terminó exitosamente y se implementa sobre la estructura LOGMIP que generó el analizador sintáctico como resultado.

La verificación semántica fue definida como una implementación del patrón VISITOR donde los elementos *"visitados"* son los componentes de la estructura LOGMIP generada por el analizador sintáctico. Esto significa que las abstracciones utilizadas para representar la gramática solo modelan la interacción de símbolos, pero no las acciones que se pueden llevar a cabo con ellos. De esta manera nuevas acciones (como el caso del chequeo semántico y, como se verá después, la generación de código LOGMIP) puedan ser incluidas en cualquier momento sin por ello modificar el modelado de la gramática.

Para poder implementar el patrón VISITOR, se incorporó el método "aceptar" el cual es el encargado de aceptar un objeto visitante y delegar la operación al método correspondiente. Además, si el objeto "aceptado" es un elemento compuesto de la gramática (un símbolo no terminal), es aquí donde se produce una aceptación de la operación sobre los componentes. Estas interacciones se pueden ver en la Fig. 6.

En este chequeo semántico, se verifica que:

- no existan declaraciones superpuestas como por ejemplo, disyunciones declaradas más de una vez, o con el nombre de un identificador de base,
- que no existan definiciones superpuestas,
- que los identificadores de base están siendo correctamente tratados, por ejemplo si las restricciones se emplean como restricciones, si los índices son correctos, si los identificadores utilizados existen en el modelo, etc.,

- que las condiciones a ser evaluadas en los términos de disyunción sean correctas y, en principio, si son mutuamente excluyentes, etc..

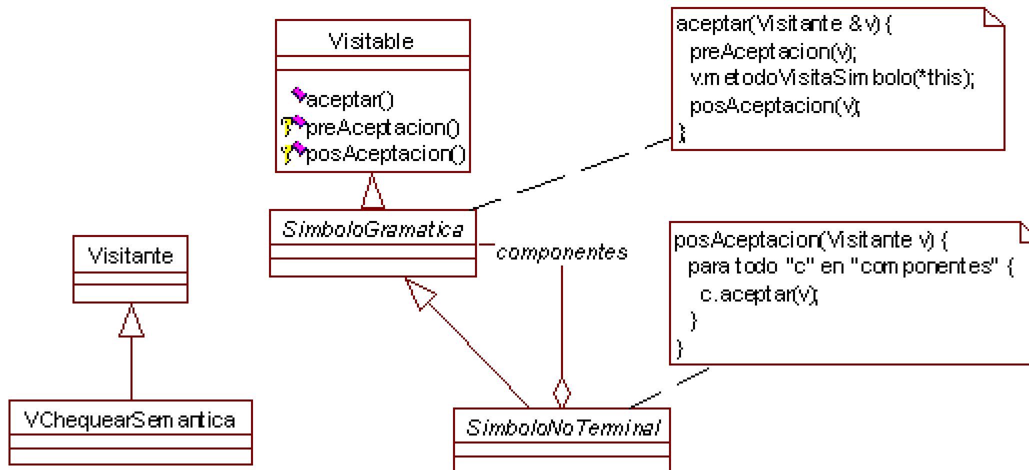

Fig 6: Implementación de VISITOR para la Verificación Semántica del Modelo LOGMIP

Si al terminar esta fase del compilador se determina que el modelo tratado es correcto semánticamente, entonces se pasa a la generación del modelo LOGMIP que se describe en la próxima sección.

Cabe notar que en caso que si en alguna parte de la validación semántica se determina que el modelo de entrada es inválido, se continua con la verificación para poder avisar al usuario la mayor cantidad de errores encontrados.

En esta etapa no se genera ningún modelo intermedio.

5.3. Generación del modelo de representación interna final (Modelo Semántico LOGMIP)

Aquí se genera un nuevo modelo LOGMIP, el cual representa el significado de la estructura generada inicialmente. En las tres primeras fases del compilador se obtiene y representa el *QUE* del modelo. Una vez verificada que la entrada es correcta, se genera un nuevo modelo que representa el *COMO* dentro del compilador LOGMIP. Esto significa que se debe traducir de una representación a otra, donde se instancian los objetos sobre los cuales luego se generan la información necesaria para el Solver LOGMIP.

Por ultimo, esta etapa es la homóloga a la generación de código objeto en un compilador tradicional, con la diferencia que ésta se representa en memoria y no en un flujo de archivo.

Esta fase también ha sido modelada como una implementación del patrón VISITOR donde la acción a llevar a cabo es que a partir de un símbolo gramatical se debe generar su correspondiente acción /representación semántica. Esto se ve en la Fig. 7. En esta figura se puede visualizar la base utilizada para representar la semántica donde se implementó el patrón de diseño *Composite*. Como resultado de esta operación del compilador, se genera el modelo semántico correspondiente a la entrada. Este modelo mantiene el conocimiento de cómo tratar la información y cual es el formato en que el Solver LOGMIP maneja esta información.

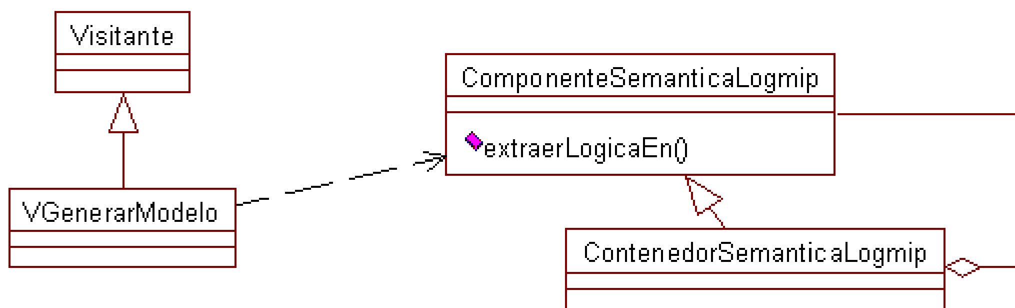

Fig 7: Implementación de Visitor para la generación de modelos semánticos LOGMIP

6. Interacciones entre el compilador LMB y LOGMIP

Para formular disyunciones se necesitan algunos identificadores del LMB. Por ejemplo, en GAMS son necesarios los identificadores de variables, ecuaciones, conjuntos y parámetros. Para tener estos identificadores disponibles en el momento de la compilación de la gramática de LOGMIP, este paso se realiza a después que el LMB a reconocido sus construcciones gramaticales y generado la tabla de símbolos. Con esto se evita que el compilador de disyunciones deba reconocer también las construcciones del LMB. Con esta estrategia se obtienen las siguientes ventajas:

- El compilador LOGMIP se puede incluir en cualquier lenguaje base, ya que debe reconocer solo la gramática de las disyunciones.
- La sintaxis y semántica del modelo matemático ya ha sido validada en el momento de la compilación de las disyunciones.
- El sistema es más fácil de mantener porque los cambios en el lenguaje base tienen un impacto mínimo en el compilador LOGMIP.

Por otra parte, el lenguaje base no necesita reconocer las disyunciones, ya que en el archivo de entrada existe una sección especial para LOGMIP que es ignorada por el compilador LMB. La Fig. 8 muestra la interacción entre ambos compiladores.

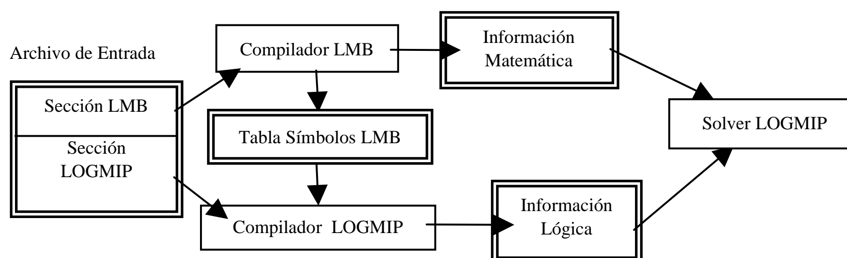

Fig. 8: Interacciones entre el compilador LMB y el compilador LOGMIP

7. Representación interna de las disyunciones e identificadores base

Para representar los identificadores que están involucrados en el modelo lógico se empleó el patrón de diseño de software ADAPTER (Gamma y otros, 1994). El objetivo del uso de ADAPTER fue nuevamente el poder minimizar la interacción entre ambos compiladores. Este patrón de software convierte la interfase de una clase en la interfase esperada por otra clase. En este caso, ADAPTER convierte la interfase del LMB en una esperada por LOGMIP. La clase *Logmip Identifier* de la Fig. 10 representa la interfase LOGMIP. Por medio de esta propuesta, si el identificador del lenguaje base cambia, sólo se debe corregir la clase *Logmip Identifier*, pero no el resto de la representación.

Notar que una disyunción está compuesta de términos y que un término a su vez puede ser una disyunción, resultando por lo tanto una estructura recursiva. Los algoritmos que se aplican a la

disyunción puede ser aplicada sobre sus términos. Las abstracciones acerca de los elementos de una disyunción y los términos que la componen se pueden ver en la Fig. 9.

Los términos explícitos del diagrama son aquellos que comienzan con IF..THEN ó ELSIF.. THEN, mientras que los términos implícitos son aquellos términos relacionados cuando la condición es negada (término ELSE). A través de estas abstracciones básicas se puede manejar la semántica asociada a las disyunciones.

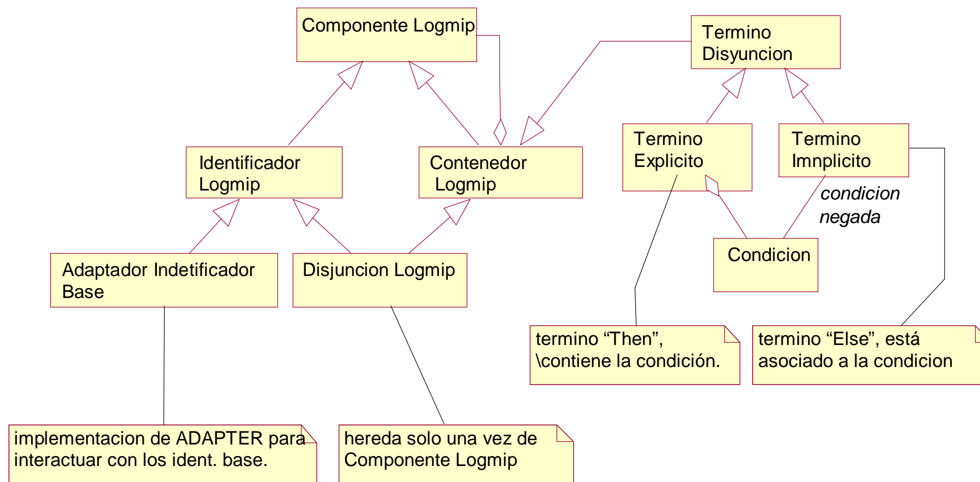

Fig. 9: Diagrama de clase de la representación interna

8. Solver LOGMIP

Una vez que la primera (compilación del LMB) y la segunda etapa de compilación (LOGMIP) ha terminado exitosamente se puede continuar con la resolución del problema.

Los posibles algoritmos por los cuales un problema disyuntivo ó híbrido pueden ser resueltos se pueden encontrar en Vecchietti y Grossmann (2000). La diferencia entre un problema puramente disyuntivo y un problema híbrido es el modo en que se formulan las decisiones discretas: solo disyunciones ó con restricciones mixtas-enteras más disyunciones, respectivamente.

El caso más general para resolverlo es convertir el modelo con disyunciones a uno mixto-entero por medio de la aplicación de la cáscara convexa de un conjunto disyuntivo, y luego resolverlo con algún algoritmo mixto-entero que el sistema base tenga. Si es lineal, después de la aplicación de la cáscara convexa el problema se puede resolver con la implementación del cualquiera método Branch and Bound, por ejemplo con los solvers OSL, CPLEX, XA. Si el problema es no lineal se puede resolver el problema con DICOPT++ que está disponible en GAMS. La Fig. 10 muestra esta situación.

Fig. 10. Diagrama de flujos en la resolución de un problema disyuntivo/híbrido

Se han resuelto diversos problemas con LOGMIP correspondientes a las áreas de diseño, scheduling, ingeniería de procesos, determinación de parámetros, síntesis, etc., los resultados obtenidos con esta propuesta en el prototipo anterior de LOGMIP pueden ser encontrados en Vecchietti y Grossmann (1999).

9. Conclusiones

En este trabajo se ha descripto la arquitectura y el modelo de representación interna de un compilador de lenguaje para problemas disyuntivos. Se seleccionó la sentencia IF..THEN.. ELSE..ENDIF por su expresividad, facilidad de uso, representación recursiva y por ser una sentencia muy conocida y usada.

Dado que la Programación Disyuntiva Generalizada incluye lógica en las formulaciones de los programas matemáticos, la implementación se realizó sobre GAMS que es un sistema para la formulación y resolución de problemas matemáticos muy conocido.

El diseño del compilador LOGMIP se basó en el empleo de una serie de patrones con el objeto de lograr independencia respecto del compilador matemático, facilidad de extensión, mantenibilidad, y eficiencia en la operación de compilación.

La representación de objetos de la arquitectura deseada (Pipes&Filters) es casi directa y permite que, una vez definidas las interfases de "comunicación" (modelos internos) el diseño y posterior implementación de cada una de ellas sea independiente de las demás. Esto permite además, la incorporación de nuevos "filters" sin afectar a la definición de los ya existentes.

Como futuro trabajo se prevé la incorporación de un generador de Modelos Gráficos, a partir del modelo inicial. Este nuevo filtro no debería modificar la configuración actual de la arquitectura, completaría el modelo existente con el agregado de uno ó más "filters".

El uso del lenguaje facilita la formulación de los problemas permitiendo al usuario escribir y modificar fácilmente el problema, de esta manera puede concentrar sus esfuerzos en generar modelos más eficientes.

Agradecimientos. Los autores agradecen a GAMS y Mitsubishi Chemicals Company por el soporte financiero al proyecto.

Referencias

- Brooke y otros, "GAMS, User's Manual", Gams Development Co., 1996.
- Darby-Dowman K., Little J., Mitra G. y Zaffalon M. "*Constraint Logic Programming and Integer Programming Approaches and Their Collaboration in Solving an Assignment Scheduling Problem*". Constraints, 1, 245-264, 1997.
- Gamma E., Helm R., Johnson R. y Vlissides J., *"Design Patterns. Elements of Reusable Object Oriented Software"*. Addisson Wesley, 1994.
- Garlan, D. y Shaw M. **"***Advances in Software and Knowledge Engineering***"**. New Jersey: World Scientific Publishing Co. (1993).
- ILOG Solver 4.3*User's Manuals.* ILOG, 1998.
- Lee S. y Grossmann I.E. *"New algorithm for Nonlinear Generalized Disjunctive Programming"*. Computers and Chemical Engineering, , 24, 9, 2125-2142, 2000.
- Raman R. y Grossmann I.E., *"Modeling and Computational Techniques for Logic Based Integer Programming"*. Comp. Chem. Eng., 18 (7), 563-578, 1994.
- Turkay M. y Grossmann I.E. ,*"Disjunctive Programming Techniques for the Optimization of Process Systems with Discontinuous Investment Costs-Multiple Size Regions"*. I&EC Research, 35 (8), 2611-2623, 1996.
- Van Hentenryck P. y Saraswat V. "*Strategic Directions in Constraint Programming*". ACM Computing Surveys, 28, 4, 701-726, 1996.
- Vecchietti A. y Grossmann I.E. **"***LOGMIP: A Disjunctive 0-1 Nonlinear Optimizer for Process System Models"*. Comp. Chem. Eng., 23, 555-565, 1999.
- Vecchietti A. y Grossmann I.E. *"Modeling issues and implementation of language for disjunctive programming"*. Comp. Chem. Eng., 24, 9, 2143-2155, 2000.
- Wallace M., Novello S. y Schimpf J.*"Eclipse: A platform for Constraint Logic Programming"*. Technical Report, IC-Parc, Imperial College, London, 1997.