ISSUES ABOUT THE DEVELOPMET OF A DISJUNCTIVE PROGRAM SOLVER

Juan J. Gil and 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

Abstract. The development of a disjunctive program solver is described in this paper. First, the implementation of a language compiler needed to formulate discrete decisions thorough disjunctions is presented. The proposed language is implemented as a superset of the commercial mathematical program system solver GAMS. The interactions between both systems are also detailed. Finally, the steps executed to solve problems formulated with disjunctions with the new system are presented.

1. Introduction

In recent years, there have been several research efforts for the inclusion of logic into mathematical programming problems. Constrained Logic Programming (CLP) and Generalized Disjunctive Programming (GDP) are examples of research areas dealing with logic in the problem formulation. The use of logic has shown advantages in the formulation and solution of optimization problems. Using logic at the problem formulation level facilitates in many cases the expression, readability and understandability of discrete decision constraints. Many discrete decisions are written in a more natural way through logic constraints and/or disjunctions. On the other hand, the solution algorithms proposed by CLP and GDP have improved the execution time needed to reach the solution compared with those generated for mathematical programs (Darby-Dorman et al., 1997, Van Hentenryck and Saraswath, 1997, Turkay and Grossman, 1996, Lee and Grossmann, 2000). Larger and complex problems can be solved with those new algorithms. Even with those advantages shown by CLP and GDP, few implementations in commercial or research codes have been made. ECLIPSE (Wallace et al., 1997) and ILOG (ILOG, 1998) are solvers developed for CLP, LOGMIP (Vecchietti and Grossmann, 1999) is a prototype for GDP. It is needed the implementation of a language for the incorporation of disjunctions and logic propositions in mathematical program solvers (Vecchietti and Grossmann 2000).

In this paper we present some issues related to the implementation of a language to express disjunctions as a superset of the commercial mathematical program system GAMS (Brooke et al, 1996). We describe the language compiler architecture and the internal representation. The solutions adopted to solve the interaction between the existing system (GAMS) and the new system (LOGMIP) are also presented.

2. Language Syntax

The reader can find in Raman and Grossmann (1994) a complete description about disjunctions, their properties and the implications into mathematical programs.

For the selection of the statements to formulate disjunctions, the following properties were considered:

- Simple syntax
- Semantic very close to the disjunction expression so that we can model blocks, which are exclusive between them.
- The syntax must be already known for a regular user.
- The statement must permit the definition of embedded disjunctions (disjunctions with terms that contains disjunctions) .

We have chosen the selection statement IF..THEN..ELSE..ENDIF because it complies with all properties previously cited.

The following notation is used to describe the syntax of the language:

Symbol	Meaning
< >	The phrase enclosed is a syntactic rule
Word	Token
[]	Optional expression
{}	Expression that can be repeated
( )	Expression enclosed can be grouped
::=	Define like
|	OR

For the disjunction expression in LOGMIP we can use the following context free grammar:

<LOGMIP Model> ::=
(<Disjunction Declaration> | <Disjunction Definition>) {**;** <LOGMIP Model>}
;

<Disjunction Declaration> ::=
disjunction identifier {**, identifier**} **;**
;

<Disjunction Definition> ::=
disjunction identifier is <If Sentence>
;

<If Sentence> ::=
if <condition> **then** <components> **else** <components> **end if ;**
| **if** <condition> **then** <components> {**elsif** <condition> **then** <components>} **end if**
;

<components> ::=
entity {**;** **entity**} [**;** <If Sentence>];

In the syntax above <components> represents constraint identifiers of the mathematical base language (MBL). Those identifiers can be represented by the MBL in different formats. By defining the components rule as sequence of entities, the purpose is to isolate LOGMIP language syntax from MBL syntax. In this way LOGMIP can extend any MBL (e.g. GAMS, AMPL, LINDO) with minimum changes in the LOGMIP language grammar.

3. Compiler Architecture

The compiler was designed using the Pipes and Filters Architecture approach (Garlan and Shaw, 1993). Two main components exist in this architecture model: a) *the filters* that perform some transformation over the information they receive and produce as output some

other information and b) *the pipes* which deal with the information transport between the filters. The LOGMIP compiler architecture based on this model is shown in Fig. 1.

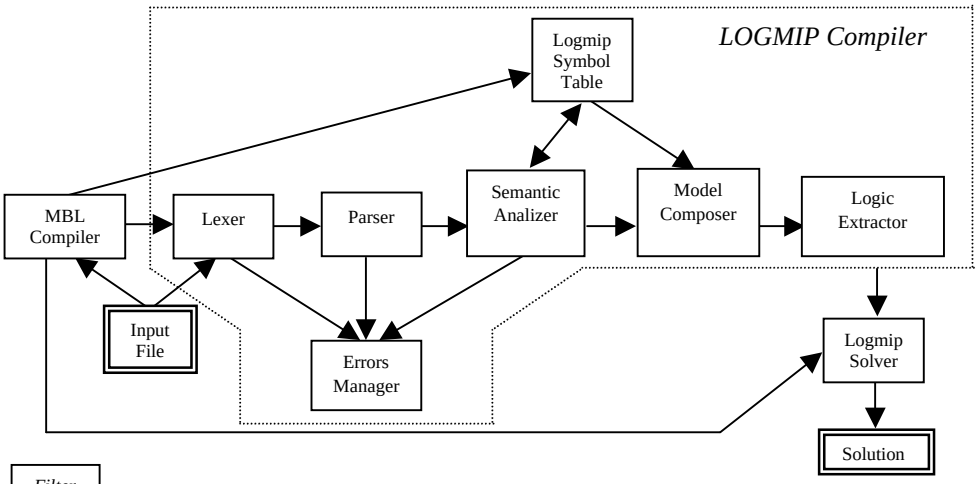

Fig1: Architecture used in LOGMIP compiler design

The following table shows the different filters composing the compiler and their action:

Filter	Action
Lexer	Transform the input into tokens that are passed to the parser
Parser	Verifies the correctness of the tokens sequence and generates an intermediate model which is passed to the semantic analyzer
Semantic Analyzer	Verifies the semantic correctness of the intermediate model. At this point all the validation about the disjunction model finish
Model Composer	Transforms the validated model in a LOGMIP disjunction representation and passes it the Logic Extractor
Logic Extractor	Takes the LOGMIP disjunction representation and generates the logic information needed by LOGMIP Solver.
Logmip Symbol Table	Receives the base symbol information and transforms them into LOGMIP identifiers
Errors Manager	Receives the information about the lexical, syntax and semantic errors encountered and display them to the modeler.

4. Interactions between MBL and LOGMIP compiler

Some identifiers of the MBL model are necessary to formulate disjunctions. For example, in GAMS language those identifiers are variables, equations, sets, and parameters. In order to know all the MBL identifiers used in the disjunction formulation, the LOGMIP grammar is checked after the MBL compiler recognizes its constructions and generates the symbol table. In this way, the disjunctions compiler avoids the recognition of the MBL symbols. With this strategy we can obtain the following advantages:

- The language can be embedded independently of the base language. LOGMIP only has to recognize the grammar of disjunctions.
- The syntax and semantics of mathematical model is already checked at the moment of the logic compilation.
- The maintainability is also easier because changes in the base language have a minimum impact in the compiler of disjunctions.

On the other hand, the base language does not need to recognize disjunctions. In the input file there exists a special section for LOGMIP which is ignored by the MBL compiler. Fig. 2 shows the interaction between both compilers.

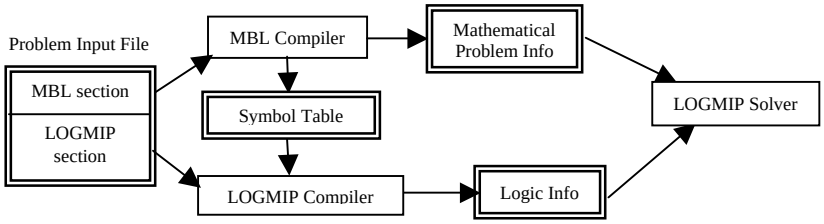

Fig. 2: Interactions between the MBL and LOGMIP compilers

5. Internal Representation of disjunctions and base identifiers

To represent the identifiers that are involved in logic model, the software design pattern ADAPTER (Gamma et al., 1994) was used in order to minimize the interaction between the MBL and LOGMIP compilers. This software pattern converts the interface of a class into an interface expected by another class. In this case, ADAPTER converts the interface of the MBL in the one expected by LOGMIP compiler. The class Logmip Identifier in Fig. 3 represents the LOGMIP interface. Using this approach, if the base identifiers interface changes, only Logmip Identifier must be corrected, but not the rest of the representation.

An identifier must be declared for each disjunction such that the modeler can define different models with the same disjunction set using the associated name.

Note that a disjunction is composed of terms, a term can be a disjunction, giving it a recursive structure. The algorithms applying to the disjunction can be applied also over its terms. The abstractions about the elements of a disjunction and the terms composing it are represented in Fig. 3.

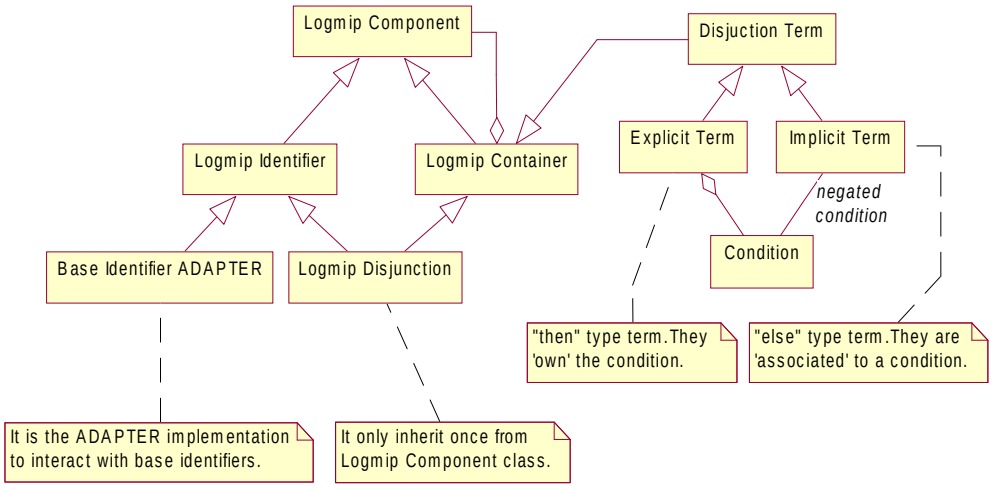

Fig. 3: Class diagram of the internal representation

The explicit terms of the diagram are those starting with IF..THEN or ELSIF.. THEN, while the implicit terms are those terms associated to implicit terms where their condition is negated. Through this basic abstractions we can handle the semantic associated to a disjunction.

6. Algorithms implementation

Once the first compilation step (the base language) and the second step (the LOGMIP language) have finished successfully we can proceed with the problem solution.

The possible paths that disjunctive or hybrid problems follow to reach the solution can be found in Vecchietti and Grossmann (2000). The difference between a pure disjunctive problem and a hybrid model is the way we are formulating the discrete decisions: with disjunctions only or involving mixed/integer constraints and disjunctions, respectively.

The problem modeled with disjunctions is converted into a mixed integer problem through the application of the convex hull of a disjunctive set, and then solved by any mathematical algorithm available in the base system (GAMS for instance). If the problem is linear, after the application of the convex hull we can solve it with the implementation of the Branch and Bound method (OSL, CPLEX, XA), if the problem is nonlinear we can solve the resulting convex hull transformation with the OA/ER/AP algorithm (DICOPT^{++}). Fig. 4 shows this situation.

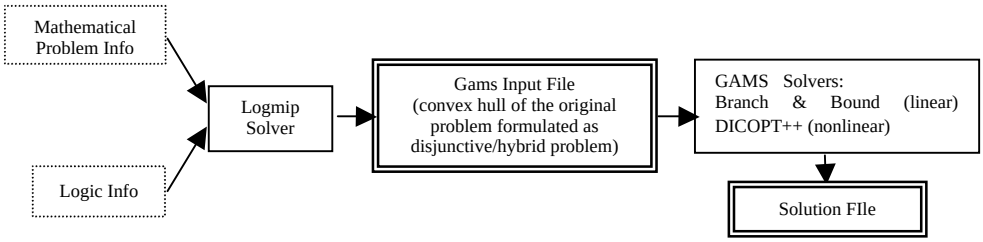

Fig. 4. Flow diagram of the problem solving

7. Examples

We have solved several examples from the process system-engineering domain with this approach. Categories of the examples cover process synthesis, batch design, spectroscopy parameters determination, design with discontinuous cost function, etc. We can not show the results obtained in the solution of these problems for space reasons. In general, lower CPU time was used to reach the solution or a better optimal solution compared with a mathematical approach was obtained. Some results obtained with this approach can be found in Vecchietti and Grossmann (2000). Examples coming from other areas have also been solved.

8. Conclusions

In this paper we have described some issues about the implementation of a disjunctive program solver. The selection statement IF..THEN..ELSE..ENDIF was selected because of its expressiveness, easy to use, recursion capability and well-known syntax and semantic. A compiler for this statement has been implemented.

Since Generalized Disjunctive Programming includes logic into mathematical program formulations, the implementation was made over GAMS which is a well known system for the formulation and solution of mathematical programs. One important objective of the implementation made was to minimize the interaction between the mathematical program system and LOGMIP. To reach this goal, we have used the software design pattern ADAPTER. ADAPTER translates the base language inputs into outputs understandable for

LOGMIP, such that if the language input changes we have to redefine ADAPTER while maintaining unchangeable the rest of the interface. The Object Oriented approach has been used to model the internal representation of the language. In this way, we can reuse the code generated for the compiler and make the system maintenance easier.

The use of the language facilitates the problem formulation making it easier to write and modify. In this way the modeler can concentrate his efforts in the generation of an efficient model instead of figuring out how to write a discrete decision with a mathematical expression.

The automatic generation of the convex hull for linear and nonlinear disjunctive models has shown advantages in the time needed to reach the solution and for some cases the optimal solution obtained is better than in a mathematical problem formulation.

Acknowledgments. The authors are grateful for financial support to GAMS Development Co. and Mitsubishi Chemicals Co.

References

Brooke et al., "GAMS, User's Manual", Gams Development Co., 1996.

Darby-Dowman K., Little J., Mitra G. and 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. and Vlissides J., "*Design Patterns. Elements of Reusable Object Oriented Software*". Addisson Wesley, 1994.

Garlan, D. and 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. and Grossmann I.E. *"New algorithm for Nonlinear Generalized Disjunctive Programming"*. Computers and Chemical Engineering, , 24, 9, 2125-2142, 2000.

Raman R. and Grossmann I.E., *"Modeling and Computational Techniques for Logic Based Integer Programming"*. Comp. Chem. Eng., 18 (7), 563-578, 1994.

Turkay M. and 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. and Saraswat V. "*Strategic Directions in Constraint Programming*". ACM Computing Surveys, 28, 4, 701-726, 1996.

Vecchietti A. and Grossmann I.E."*Modeling issues and implementation of language for disjunctive programming*". Comp. Chem. Eng., 24, 9, 2143-2155, 2000.

Wallace M., Novello S. and Schimpf J."*Eclipse: A platform for Constraint Logic Programming*". Technical Report, IC-Parc, Imperial College, London, 1997.