Using design patterns for a compiler modeling for posing disjunctive optimization programs

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

Abstract. In this article, the use of software design patterns for modeling the stages of a compiler is presented. The compiler is generated for expressing disjunctions and logic propositions into optimization mathematical programs. It works linked to a mathematical modeling system, because the introduction of logic into those programs completes them instead of replacing them. It works as a post-compilation step of the mathematical language compiler. The language for disjunctions and logic propositions is based on the proposal of Vecchietti and Grossmann (2000). In order to accomplish the objectives: independence of the mathematical system, flexibility for introducing changes and easy to maintain; several software design patterns are used such as: *Visitor*, *Composite*, *State* and *Adapter*. The compiler is now linked to the mathematical program system GAMS. Several test are performed to check its behavior. In the future some other mathematical systems will be used to link the compiler.

1. Introduction

Traditionally, optimization models involving linear/non-linear equations and constraints, and also discrete decisions are represented as mixed integer non-linear program problems (MINLP). These problems are difficult to solve, one important issue to reach the solution is to provide an efficient model for the discrete decisions. Over the past five years there was an intensive research activity on disjunctive programming as an alternative for the MINLP formulation. Disjunctions and logic propositions are used to represent the discrete decisions in the continuous and discrete space respectively. The logic is introduced at the level of the problem formulation and solution techniques. The main research areas dealing with logic into mathematical program problems are: Disjunctive Programming (Raman and Grossmann, 1994; Turkay and Grossmann 1996, Bjorkqvist and Westerlund, 1999; Vecchietti and Grossmann, 1999) and Constrained Logic Programming (Hajian et al. 1995; Darby-Dowman et al, 1997). Since the modeling framework proposed by Raman and Grossmann (1994), new algorithms and solution techniques have been proposed. One of main reasons those techniques have not been widely used yet, is because there is not a system to write those type of models. The most known systems for solving mathematical optimization problems, e.g. GAMS, LINDO, AMPL, are not prepared for posing a disjunctive model. They do not have a language for expressing disjunctions and logic propositions. Therefore, it is necessary a language to incorporate disjunction and logic propositions in to mathematical program problems. This work is concerned about disjunctive models and the language compiler implementation for writing this type of models based on the language approach proposed by Vecchietti and Grossmann (2000). The compiler has been implemented in a computer code called LogMIP. This article describes the compiler stages, their modeling and design using software patterns, that provide to the compiler several important features such as: modularity, flexibility, maintainability.

2. Compiler stages

Compilers by definition take a string as input and produce another string as output. Text formatters, programs that convert file formats or different programming languages drop in the category of compilers. One of the main lessons learnt about compilers is how to split it into parts. At the highest level there are three parts: the *front end* that understands the syntax of the source language, the *mid-end* that performs high level transforming/optimizations and the *back end* that produces the output in a previously established language. Fig. 1 represents this situation.

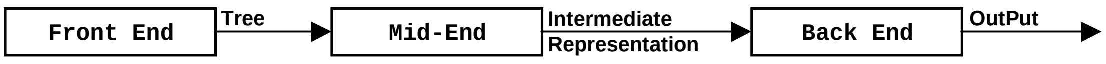


Fig. 1: Parts of a compiler

At a lower level, a compiler consists of various stages (Aho et al., 1986). In the LogMIP compiler designed five of them are implemented: Lexer, Parser, Semantic Analysis, Intermediate Code extractor and Code Extractor. Each of these stages will be explained in this paper. In Fig. 2 it is shown this fives stages and the data structure and data passed between these stages.

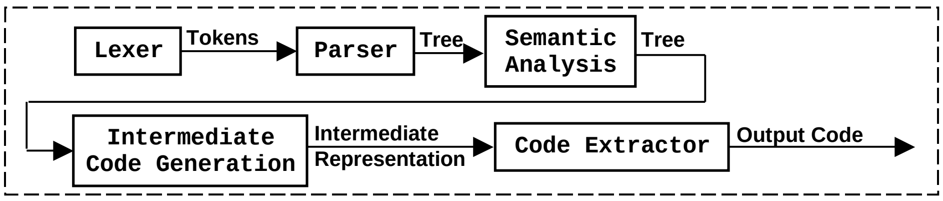


Figure 2: Phases adopted in LogMip's Compiler

For LogMIP we are extending an existing language for expressing mathematical program problems, adding to it capabilities for expressing disjunctions and logic constraints. This situation makes LogMIP Compiler (LMC) has special characteristics, because the interaction between both compilers must be solved. This situation can be seen in Fig. 3:

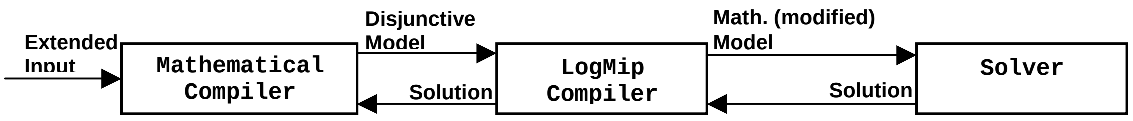


Figure 3: LogMip Compiler and Mathematical Compiler interactions

From Fig. 3, it can be seen that LMC is receiving a mathematical program model syntactically correct according to the mathematical language. The Extended Input file (with mathematical and disjunctive constraints) has been checked in its mathematical constructions. Then the LogMIP compiler checks the logic constructions and, if they are correct, the output passes to the Solver such that the model is solved. Using this approach a great level of independence is obtained, because the LMC can extend any mathematical program system. Besides, on the other end any Solver implementing the solution algorithms can be used.

The following sections explain how is reached the first step, how is mapped the second one, shows the capabilities of this approach and the future work in LogMIP Compiler.

3. Design Patterns used into LogMIP compiler

Lexer (or lexical analyzer)

The lexer is the first step the language compiler. Its purpose is to decompose the input stream into *tokens*, which represents reserved words in the language under analysis (LogMIP and Mathematical Language) and some string that have extra information associated (e.g.: identifiers, numbers). Besides, each token have a position in the input stream and a code that identifies it from the other tokens of the language. Fig 4. shows those concepts in an object oriented view:

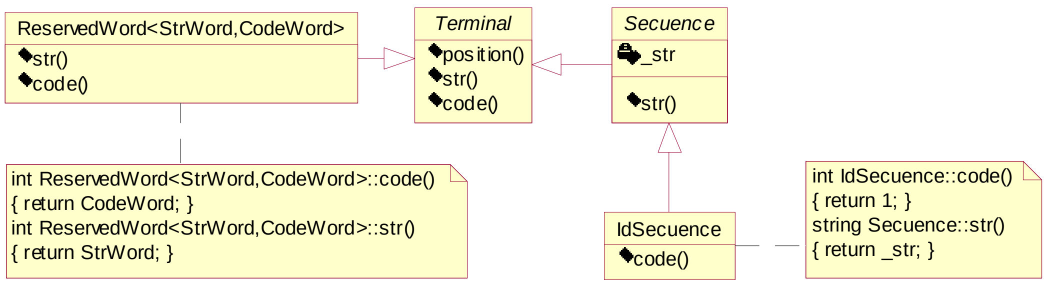


Figure 4: Terminal Symbols Representation in LogMip Compiler Lexer.

Parser (or Syntax Analyzer).

Parsing is the process of understanding the language syntax, such that it can be represented by the compiler internal data structures. The most sophisticated ideas humans can relay to computers are communicated with programming languages. This has made programming languages compromises between the human thought process, the computers execution process and the computers capability to understand a language. The parser deals with the last facet of the problem. In a more strict definition, the parser is the one, which verifies if the input is valid under the grammar describing the language. If the input is a valid one, the parser must generate a syntax tree representing in an internal structure the information modeled on it. This syntax tree is an intermediate representation of the input analyzed, which is used by the next phases of the compiler to get a result. Usually it consists of an n-tree where the leaves are tokens (terminal symbols) found in the input, the intermediate nodes represent the different rules matched by the parser, and the root symbolizes the input itself. For the tree representation is used the *Composite* design pattern (Gamma, 1994). This pattern composes objects into tree structures to represent part-whole hierarchies, and lets clients treat individual objects and compositions of objects uniformly. The implementation is shown in Fig. 5.

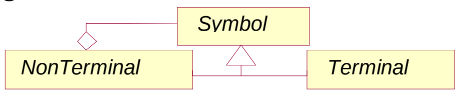


Figure 5: Syntax Tree with Composite

The tree is created as follows: first, the parser always has a node representing the root (the input representation). Then, once the parser recognizes a rule, it creates a new node (a symbol) that is the internal representation for that rule and adds it to the tree. Because this process consumes huge amounts of memory and processor's time, it is avoided the creation of nodes and relationships when it is known the tree will not be used. It occurs when the parser founds a lexical or a syntax error. In this case the compiler will not attempt to continue with the other phases because it knows the input is incorrect. In this situation the compiler will try to find more lexical/syntax errors on the input. Once the parsing is done, no more phases will then be executed. This means that while the input is valid the parser must construct nodes and add it to the syntax tree. When a lexical or syntax error is discovered, the parser stops the tree generation because the compiler will finish the execution after this phase. The parser state diagram is shown in Fig. 6.

The *State* design pattern (Gamma, 1994) is used to model this situation. The abstraction that encapsulates the parser states and their behavior is *AbstractSyntaxTreeConstructor* where two specializations are introduced, which are *AstcValid* for the "constructor" state and *AstcInvalid* for the errors finder. Under this situation it can be considered that there are two families of symbols:

Real Symbols and Null Symbols. Real Symbols are those instantiated while the input is valid. Null Symbols are used for invalid inputs. The creation of those object families (Real and Nulls) is performed by the *Abstract Factory* pattern implementation.

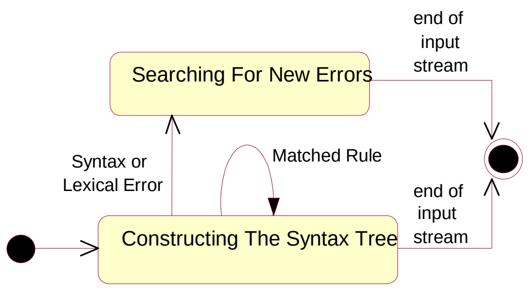
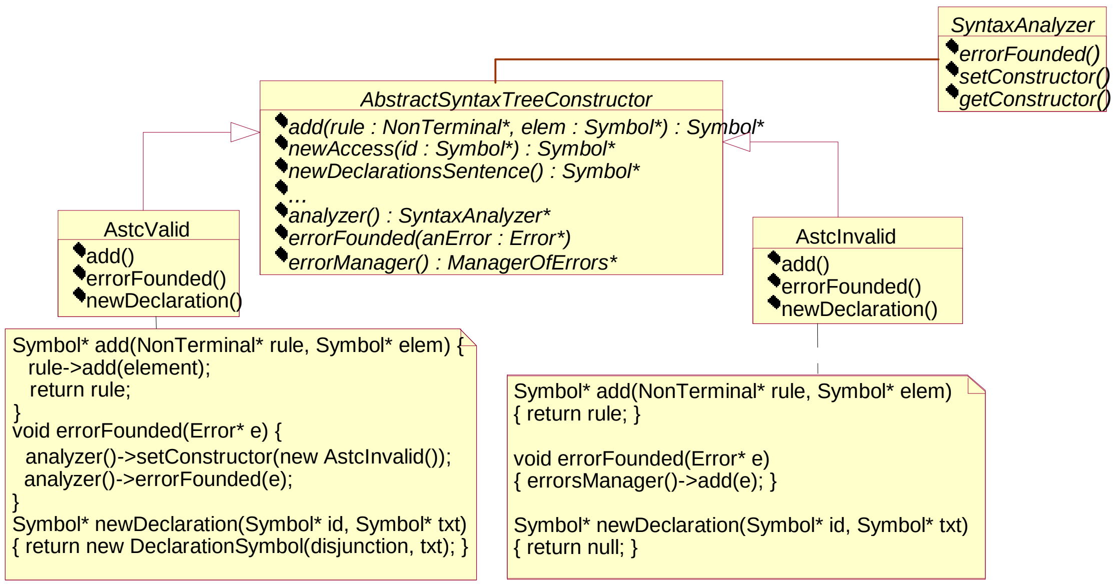


Figure 6: States of the LogMip Parser

With this approach, the parser could change the state and still continue the process of creating symbols (nodes) for the syntax tree. More independence respect of the Subjacent Mathematical Compiler (SMC) is obtained because this representation permits to change the leaves (terminal symbols) or even the intermediate nodes (the rules that conforms the LogMip grammar) without changing the parser itself (note that this change could be necessary to make LogMIP grammar naturally absorbed by the SMC grammar).

The difference between the two specializations is that while *AstcValid* create nodes and the relationships between the nodes, *AstcInvalid* does not make anything about them. Fig. 7 shows the interface of *AbstractSyntaxTreeConstructor* and the implementations of three methods: one that relates two symbols, one that creates a symbol, and the other that manipulates an input in the specialization.



SyntaxAnalyzer
errorFounded()
setConstructor()
getConstructor()
AbstractSyntaxTreeConstructor
add(rule : NonTerminal*, elem : Symbol*) : Symbol*
newAccess(id : Symbol*) : Symbol*
newDeclarationsSentence() : Symbol*
...
analyzer() : SyntaxAnalyzer*
errorFounded(anError : Error*)
errorManager() : ManagerOfErrors*
AstcValid
add()
errorFounded()
newDeclaration()
AstcInvalid
add()
errorFounded()
newDeclaration()

```
Symbol* add(NonTerminal* rule, Symbol* elem) {
  rule->add(element);
  return rule;
}
void errorFounded(Error* e) {
  analyzer()->setConstructor(new AstcInvalid());
  analyzer()->errorFounded(e);
}
Symbol* newDeclaration(Symbol* id, Symbol* txt)
{ return new DeclarationSymbol(disjunction, txt); }
```

```
Symbol* add(NonTerminal* rule, Symbol* elem)
{ return rule; }

void errorFounded(Error* e)
{ errorsManager()->add(e); }

Symbol* newDeclaration(Symbol* id, Symbol* txt)
{ return null; }
```

Figure 7: State & Abstract Factory Patterns in the LogMip Abstract Syntax Tree Constructor

The Semantic Analyzer.

This is the first step of the denominated *mid-end* part. This analyzer is the responsible for evaluating if the problem input stored on the syntax tree (sentences, expressions, etc.) is conformant with the semantics of the input language. At this point LogMIP Compiler evaluates:

- ➢ the declarations of disjunction entities,
- ➢ if the modeler is reinterpreting the mathematical identifiers,
- ➢ if the modeler is declaring a disjunction that is already declared,
- ➢ if the modeler is defining a disjunction more than one time,
- ➢ if an access to an identifier corresponds to the category expected (LogMIP and Mathematical ones)
- ➢ and the relationships between the disjunctive term conditions.

At this point it is important to define the interaction between de LogMIP compiler (LMC) and the Mathematical compiler (SMC). The interaction occurs at the level of identifiers, because identifiers of the mathematical language are used to define constructions in LogMIP language. Two implementations of the *Adapter* design pattern are used for this purpose: one is defined for interact with all the entities (*Identifier abstraction*), and the other is modeled for treating mathematical identifiers loader (*IdentifiersLoader* abstraction). In this way, if LMC is linked to a different SMC, a new implementation of *Identifier* must be done plus a new loader of SMC identifiers table. No other change is necessary on the rest of the compiler. Fig. 8 shows the *Adpater* implementation and how the *Template Method* pattern is used to define a common algorithm to load the mathematical symbols. This method is used to guarantee that when a mathematical symbol is loaded all the conditions to access it are already loaded.

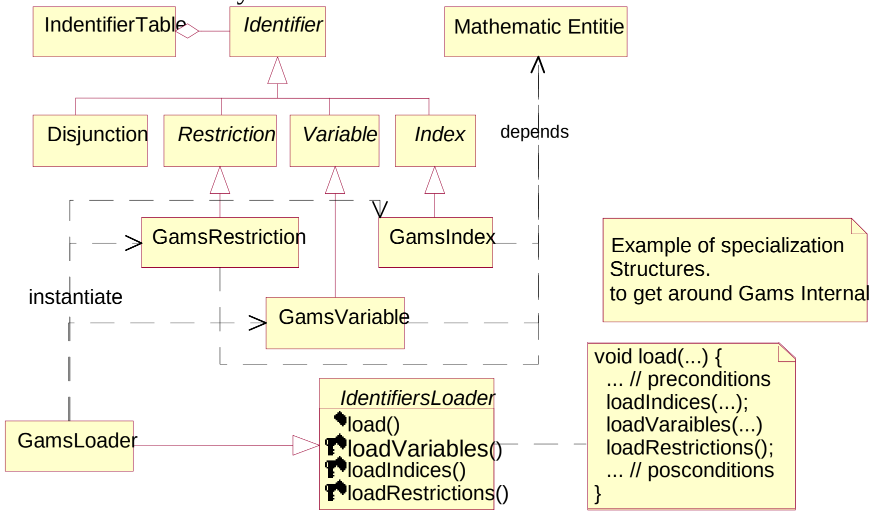
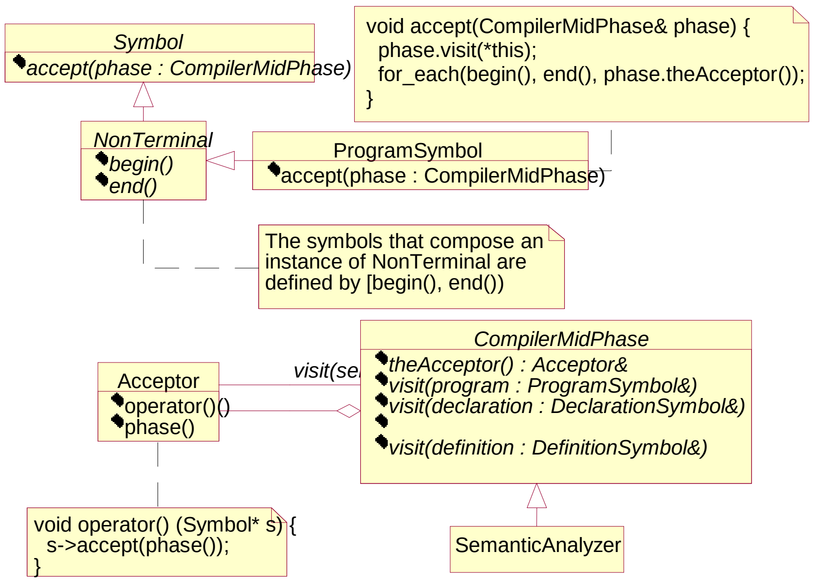


Figure 8: Adapter & Template Method Patterns in the LogMip Compiler

The *Visitor* design pattern is used to implement the *mid -end* algorithms that visit the nodes of the syntax tree. This choice was made because this pattern allows the algorithm definitions to operate over the abstractions without modifying them. Using this approach the compiler could add and subtract operations on the syntax tree without modifying its representation. Moreover, the *NonTerminal* abstraction (the ones who represents the different rules that conforms the LogMIP grammar) has been modeled with an implementation of the *Iterator* method, such that *Visitor* can parse non-terminal symbols without known its internal structure. Complement this approach an implementation of the *Command* design pattern, which lets the compiler to use the iteration algorithms over the tree, encapsulating the invocation to the *accept* method. This model is shown in Fig. 9:


Symbol
accept(phase : CompilerMidPhase)
NonTerminal
begin()
end()
ProgramSymbol
accept(phase : CompilerMidPhase)
The symbols that compose an instance of NonTerminal are defined by [begin(), end())
CompilerMidPhase
theAcceptor() : Acceptor&
visit(program : ProgramSymbol&)
visit(declaration : DeclarationSymbol&)
visit(definition : DefinitionSymbol&)
Acceptor
operator()()
phase()
visit(se
SemanticAnalyzer

```
void accept(CompilerMidPhase& phase) {
  phase.visit(*this);
  for_each(begin(), end(), phase.theAcceptor());
}
```

```
void operator() (Symbol* s) {
  s->accept(phase());
}
```

Figure 9: Visitor & Command Patterns at LogMip Syntax Tree/Mid-End

When the parser finds an access to an identifier, it adds a new node in the syntax tree that is an instance of the *AccessSymbol* class. This class represents any access in the model, access to LogMIP disjunction identifiers, mathematical identifiers (variables, restrictions, indices) or individual index items. Due to there is only one method that “parse” a node, the semantic analyzer must consult itself the access type is validating, which can be deduced from the context information. After that, it performs the corresponding action. This situation limits the analyzer behavior to a numbered of access types, meaning that if a new type of access or a new type of context access increases LogMIP semantics, the analyzer must be changed to consider the new situation. To solve this problem, the semantic of the accesses was considered as a part of the

Semantic Analyzer states. The design pattern *State* was implemented to model that behavior because it allows new states (access types) without introducing changes in the Semantic Analyzer.

In this way, when the analyzer arrives to an access symbol, delegates the corresponding actions to the "access semantic state". This is shown in Fig. 10.

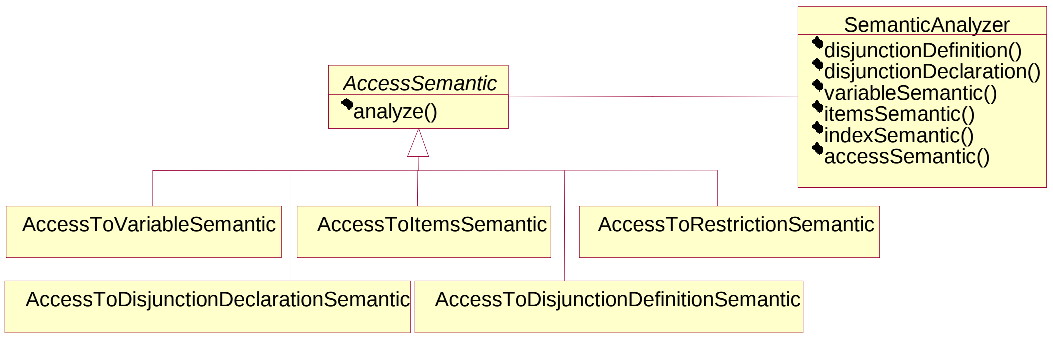


Figure 10: State Pattern Design in LogMip Semantic Analyzer

<u>*Intermediate Code Generation*</u>.

This phase is the responsible of converting the human legible representation into a machine code representation. At this point all language constructions have been checked, they are free of errors, and the compiler can now traduce the input to a more efficient representation, not necessary the final one, introducing some improvements (called optimizations). In our case we have to iterate over the syntax tree to generate a new interpretation of the logic information modeled by disjuntions and logic constraints. This phase is related to the *mid-end* part and is modeled using the *Visitor* implementation, which has been introduced with the Semantic Analyzer. This phase uses also the *State* design pattern already explained for the semantic accesses. So that, in this section it will be explained the basics of the *Intermediate Code Generator* (ICD) (see Fig. 11).

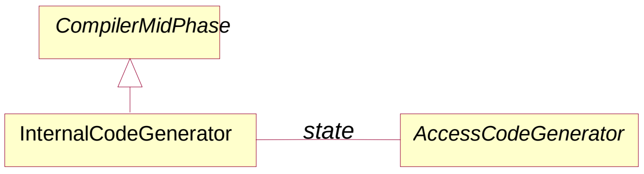


Figure 11: Internal Code Generator abstraction in the

The internal code representation is similar to the nodes for the syntax tree viewed before. In deed, there is a one to one relationship for almost every symbol that is non-terminal. To model this semantic representation a mix between ***Visitor & Composite*** patterns was used (Fig. 12).

In the *InternalSemantic* abstraction, there are three methods defined: *accept()* defines new operations over the ICR without changing it, *generateLogic()* insert into the output stream the information about LogMIP constructions (the disjunctive logic) in a pre-established format; and finally *generateLogicUsing()*, which is used to insert into the output stream LogMIP constructions with some restrictions in the generated logic. The graph of this model is shown in Fig. 12.

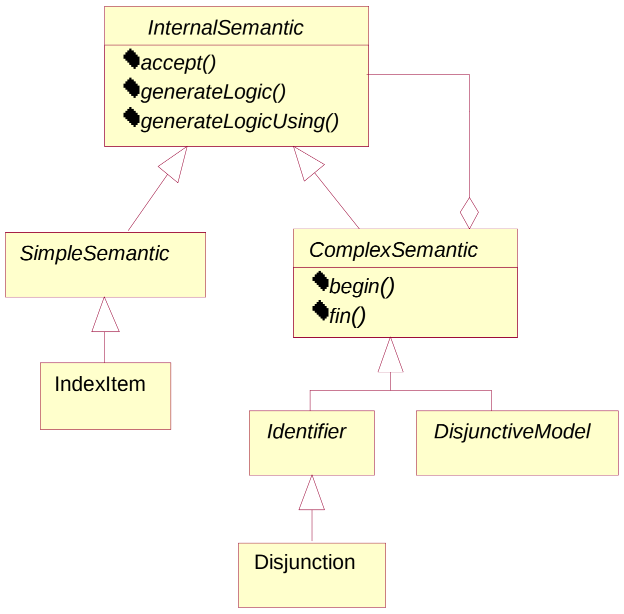


Figure 12: Internal Code representation for the LogMip's

Another abstraction called *CompilerBackPhase* is defined to visit *InternalSemantic*'s specializations, but it is not used at this moment of the model, it is only defined to allow the expansion of LMC such that it adopt some new action without change the semantic abstractions.

4. The Compiler Abstraction

In the previous sections the components and the internal representation of the LMC have been presented, but nothing has been said about the compiler itself. Although, the compiler can be defined by using its components (lexer, parser, table of identifiers and so on) without nothing more, it would be complex to see all the interactions and relationships existing between them. Further more, it could be too complex to define a new specialization of LMC to interact with another mathematic compiler.

To encapsulate this knowledge and to provide a single, common interface to the compiler, the *Facade* design pattern is used. By adopting it, a compiler client only has to request the compilation of the input. If a new mathematic compiler will be linked to LMC, it will be necessary the specialization of the concept given before in some pre-defined way. The *Facade* class diagram is shown in Fig. 13.

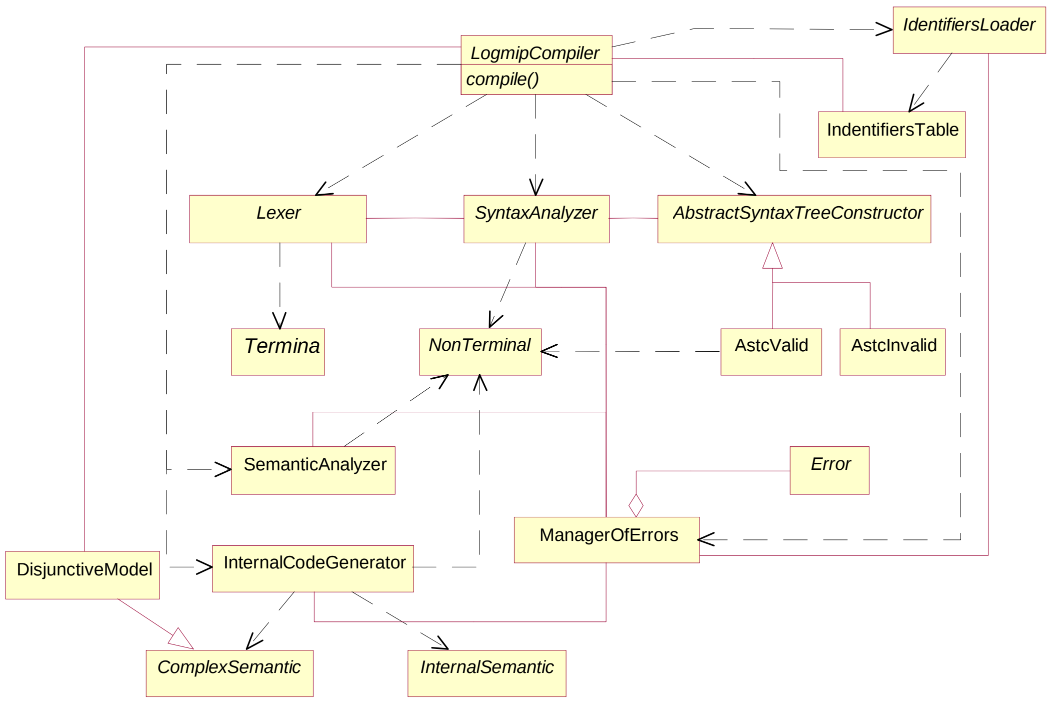
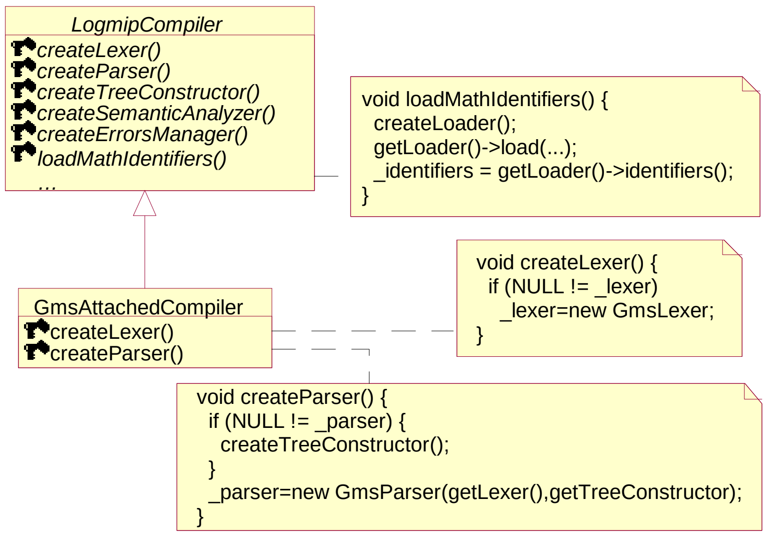


Figure 13: Facade Design Pattern in LogMip's Compiler

Besides, the compiler has to permit a new specialization to achieve new mathematical compilers. Such objective has been traduced in the introduction of the *Builder* design method. With this, a modeler can introduce a new specialization of *LogmipCompiler* without worries about how the compiler has to deal with them. The *Builder* implementation is presented in Fig. 14. In that figure is also shown a possible specialization of *LogmipCompiler* to interact with the mathematical system GAMS.

LogmipCompiler abstraction provides a common sequence to perform the compilation of the inputs leaving to its specialization how the actions of this sequence are executed. This is modeled using the *Template Method* pattern over the *compile()* method of *LogmipCompiler*. The algorithm used as the implementation of the template method is shown in Text 1.

An abstraction denoted as *Arguments* provides the independency about the way LMC can be executed. This means that the specialization of *LogmipCompiler* could define extra invocation parameters, which do not affect *compile()* method. If a mathematic compiler needs something not provided yet this model can insert it into the specialization. The arguments abstraction is an *Abstract factory* absorbing the parameters needed by the specialization.



```
void loadMathIdentifiers() {
  createLoader();
  getLoader()->load(...);
  _identifiers = getLoader()->identifiers();
}
```

```
void createLexer() {
  if (NULL != _lexer)
    _lexer=new GmsLexer;
}
```

```
void createParser() {
  if (NULL != _parser) {
    createTreeConstructor();
  }
  _parser=new GmsParser(getLexer(),getTreeConstructor);
}
```

Figure 14: LogMip compiler as an Builder pattern implementation

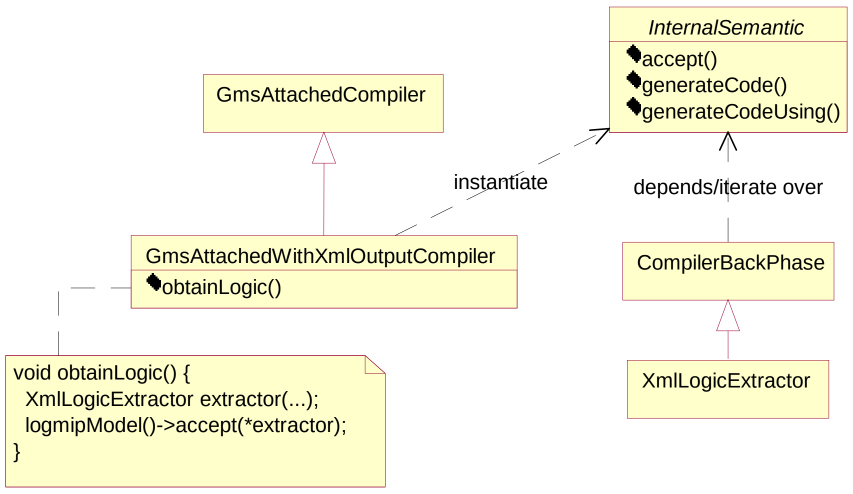
```
void LogmipCompiler::compile(Arguments& arguments) {
    createErrorsManager();
    precompile(arguments);
    analyseSyntax();
    analyzeSemantic();
    generateCode();
    obtainLogic();
    solve();
}
```

Text 1: Template Method pattern in the LogMip's Compiler

Finally, note that the code extraction phase of the LMC it is not explained anywhere. The reason is because its definition has been delegated to the *generateLogic()* method defined a*t InternalSemantic*. This is shown in Text 2. This is not a limitation in the code extraction phase. If a solver does not understand the output of the logic generated by LogMIP compiler, a new Code Extraction Phase can be generated as a specialization *of CompilerBackPhase*. Fig. 15 shows an example of this.

```
void LogmipCompiler::generateCode() {
    logmipModel()->generateLogic(outputStream());
}
```

Text 2: Template Method pattern in the Code Extractor Phase



```
void obtainLogic() {
  XmlLogicExtractor extractor(...);
  logmipModel()->accept(*extractor);
}
```

Figure 15: A possible instantiation of CompilerEndPhase, a logic extractor in XML format.

5. Conclusions

We have presented the model of a compiler to express disjunctions and logic propositions with unusual characteristics because it works as a post-compilation step. The input of LMC is the output of a mathematical compiler, and the output of LMC is expected by another application commonly a mathematical/logic solver. Under this context, the objectives for the LogMIP compiler generation were: a high independence from the mathematical compiler that LMC is extended, flexible to introduce new constructions and easy to maintain.

To accomplish those objectives we have used some special software design patterns: the *State* allows as to extend the model in an easy way by encapsulating the possible states of the abstractions and to change the different states the abstractions can hold; *Visitor* permits a flexible model extension by defining the operations in the mid-end and back-end stages out of the internal compiler representation, *Adapter* allows the independence of LMC from the SMC. Other software design pattern as was explained before helps in accomplish the objectives of this work.

LMC is now linked to the mathematical system GAMS, we are testing its behavior. We are working on the other end of the compilation: the solvers. For the future the plan is to link LMC to other SMC like AMPL

References

- Aho, A; Sethi R. and Ullman, J., “Compilers: Principles, Techniques and Tools”. Addison-Wesley, 1986.
- 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.