Graph-Based Optimization Modeling Language

Auteur : Miftari, Bardhyl
Promoteur(s) : Ernst, Damien
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en informatique, à finalité spécialisée en "intelligent systems"
Année académique : 2020-2021
URI/URL : http://hdl.handle.net/2268.2/11231

Master in applied sciences
Computer science and engineering
Year 2020-2021

Master Thesis

Graph-Based Optimization Modeling Language

Master thesis submitted for obtaining the Master's degree in Computer science and Engineering

Academic Supervisor
ERNST Damien
Institution
University of Liege

Author
MIFTARI Bardhyl

Liège

First Semester 2020-21

Acknowledgements

During the writing of this thesis, I received a lot of support and assistance without which this work would not have been the same.

First, I would like to thank Professor Damien Ernst for the opportunity he gave me to work on such a great project, for his supervision and counseling. I also would like to thank everyone at the lab for welcoming me. I thank Hatim Djelassi for his feedbacks on this work. A special mention must go to Mathias Berger and Adrien Bolland who helped me through-out this thesis, I want to thank you for your patience, the discussions we had, your valuable guidance and all your feedbacks. Thank you.

In addition, I would like to thank my family and everyone who helped me. I must mention Ilir Miftari, Joachim Roekens, Olivia Dulon, Corinne Schleich, Oceane Rumfels and Philippe Schneider who all played their part in important discussions as well as happy distractions.

Abstract

Mathematical optimization has come to play a key role in numerous disciplines in recent years. In this work, we focus on a class of problems involving the optimization of linear discrete-time dynamical systems over a finite time horizon and possessing a natural block structure. Such problems arise in a number of fields, including energy systems planning and supply chain management. The typical workflow of optimization practitioners includes four basic steps, namely formulating the model, encoding it in a computer, solving it and post-processing it. The dominant approach for the second step makes use of algebraic modeling languages (AMLs), which make it possible to write problems in a form close to the typical mathematical notation. However, AMLs are usually ill-suited for exploiting the block structure that a problem may display. A second approach, the object-oriented modeling languages (OOMLs), possess a block structure implementation of problems but lacks of the easy mathematical encoding. To alleviate this, we design and implement a language, named the Graph-Based Modeling Language (GBOML), that natively supports the definition of problems with such structure, and exploits it to facilitate their encoding and post-processing. GBOML also possesses a formulation close to the mathematical one. This language can be viewed as a hybrid language, somewhere between AMLs and OOMLs. In addition, we implemented a method to retrieve a solution for problems encoded in GBOML. In this work, we formally introduce GBOML, explain its implementation and demonstrate its usefulness with an energy system planning example.

Contents

Chapter 1

Introduction

Mathematical optimization, which provides a rigorous framework and methods for selecting the best element(s) among a set of candidates based on some pre-defined criterion, has come to play a key role in a variety of disciplines in recent years, ranging from biology to machine learning. To illustrate this point, it is worth reviewing a couple of fields of application. In the context of energy systems, Berger et al. [1] proposed an optimization-based framework to tackle long-term centralized planning problems of multi-sector, integrated energy systems including electricity, hydrogen, natural gas, synthetic methane and carbon dioxide. In computer networking, Low et al. [2] developed an optimization-based approach for flow control problems where the goal is to maximize the aggregate source utility over their transmission rates. In genetics and biology, optimization methods exist for the reconstruction of phylogenetic networks [3] and parametric alignment of sequences [4] (i.e. problem of computing the optimal-valued alignment of two DNA sequences given the edit weights to turn one string to another). In supply chain management, Choi et al. [5] proposed a mathematical optimization model that helps solve real-life business chain decisions problems such as optimizing the location and size of warehouses. Optimization methods were used in fields as diverse as urban safety. Indeed, in [6], a model to find the optimal evacuation routes (i.e. fastest route to evacuate the whole population) in the event of a disaster is discussed. Many more examples and application areas can be found [7–9], which illustrate the versatility and usefulness of optimization in engineering and science.

An optimization model is an abstract mathematical representation of a problem where one seeks to find the best outcome among a set of alternatives. The basic ingredients required to construct a model are a set of variables, a set of parameters, an objective function and a set of constraints. Parameters represent quantities that are known *a priori*, that is, they store the input data to the problem. Variables, on the other hand, represent quantities that are unknown before the problem is solved. Assigning values to the variables yields a candidate solution to the problem. Constraints define the range in which variables can take their values, as well as the relationships between different variables and parameters. In other words, they define the set of acceptable solutions (that are usually called *feasible*). Finally, the objective function expresses the criterion used to evaluate how good a given solution is. Solving the problem therefore consists in finding a set of values for the variables that maximizes or minimizes the

objective function while satisfying all constraints.

When working with optimization models, the typical workflow includes (at least) four basic steps, namely formulating the model, encoding it in a computer, solving it and post-processing it. First, model formulation involves the specification of variables and parameters as well as the definition of the objective function and the constraints in order to obtain a compact mathematical representation of the problem at hand. Second, we encode the corresponding problem in a format that is intelligible by a computer, typically using a so-called modelling language. Third, we make a link from the computer representation to a solver and solve the problem. Lastly, we post-process the results returned by the solver.

This work focuses on the second step of the optimization workflow. More precisely, we are interested in providing a way to encode a particular class of linear optimization problems involving discrete-time dynamical systems and possessing a block structure that can be encoded by a sparse connected graph. Problems with such structure naturally arise in a number of fields, e.g. in energy systems planning [1] and supply chain management problems [5]. Unfortunately, typical modeling languages are ill-suited for exploiting the underlying problem structure in a useful way. To alleviate this, we design and implement a novel optimization modeling language called the Graph-Based Optimization Modeling Language (GBOML) that exploits this structure in order to facilitate the encoding and the post-processing of these problems.

This document is structured as follows, we first define the class of problems we wish to model. Second, we review the relevant literature, the main modeling languages and highlight their shortcomings with respect to our needs. Third, we provide the theoretical background necessary for fully understanding the solution we propose. Fourth we formally introduce GBOML. We then present our implementation for representing GBOML in computer memory and a way to solve problems expressed in our language. We conclude this work with an application of the language to an energy system planning, present a conclusion and discuss future work avenues.

Chapter 2

Problem Definition

Chapter abstract: *This chapter formalizes the problem tackled in this work.*

We consider a particular class of problems involving the optimization discrete-time dynamical systems over a finite time horizon and possessing a block structure that can be encoded by a sparse connected graph. In this setting, the time horizon is discretized into a set of time periods and each node has its own set of variables, constraints and a local objective function describing its contribution to the system-wide objective. The edges between the nodes express the relationships between the nodes. The variables in each node are split into a set of input, internal and output variables. Each variable is defined for the whole optimization horizon. Linking two nodes enforces the equality between an output variable of the first node and an input variable of the second for each time period. Note that this class of problems naturally appears in a number of fields and have already been considered elsewhere [10].

More formally, let $\mathcal{N}, |\mathcal{N}| = N$, be a set of nodes and let $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ be a set of edges. Let us denote by $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ the graph induced by the aforementioned node and edge sets. Let us denote by $\mathcal{T}, |\mathcal{T}| = T$, the set of time periods considered. In this thesis, we are interested in optimization problems whose block structure can be readily encoded by a sparse connected graph (i.e., such that $|\mathcal{E}| << N(N-1)/2$). Then, each block represents a subproblem with input, internal and output variables $U_n^+ \in \mathcal{U}_n^+ \subseteq \mathbb{R}^{d_n^+ \times T}$, $X_n \in \mathcal{X}_n \subseteq \mathbb{R}^{d_n \times T}$ and $U_n^- \in \mathcal{U}_n^- \subseteq \mathbb{R}^{d_n^- \times T}$, where d_n^+, d_n and d_n^- are the number of input, internal and output variables defined per time period at node $n \in \mathcal{N}$. In other words, each row of U_n^+, X_n and U_n^- is a vector variable whose entries correspond to a specific time period. In addition, we denote by $h_n^{\leq} : \mathcal{X}_n \times \mathcal{U}_n^+ \times \mathcal{U}_n^- \mapsto \mathbb{R}^{c_n^{\leq}}$ and $h_n^{=} : \mathcal{X}_n \times \mathcal{U}_n^+ \times \mathcal{U}_n^- \mapsto \mathbb{R}^{c_n^{=}}$ the functions used to define inequality and equality constraints at node $n \in \mathcal{N}$, where $c_n^{\leq}$ denotes the number of inequality constraints and $c_n^{=}$ is the number of equality constraints. Finally, we denote by $f_n : \mathcal{X}_n \times \mathcal{U}_n^+ \times \mathcal{U}_n^- \mapsto \mathbb{R}$ the local objective function at node $n \in \mathcal{N}$, and $g_e^{=} : \mathcal{U}_n^+ \times \mathcal{U}_n^- \times \mathcal{U}_m^+ \times \mathcal{U}_m^- \mapsto \mathbb{R}, e = (n, m) \in \mathcal{E}$ a set

of functions defining the topology of the system. The full problem is given by the equations:

$$\begin{aligned} \min \ & \sum_{n\in\mathcal{N}} f_n(X_n, U_n^+, U_n^-) \\ \text{s.t. } & h_n^{\leq}(X_n, U_n^+, U_n^-) \leq \mathbf{0},\ \forall n \in \mathcal{N} \\ & h_n^{=}(X_n, U_n^+, U_n^-) = \mathbf{0},\ \forall n \in \mathcal{N} \\ & g_e^{=}(U_m^+, U_m^-, U_n^+, U_n^-) = 0,\ \forall e = (n,m) \in \mathcal{E} \\ & X_n \in \mathcal{X}_n, U_n^+ \in \mathcal{U}_n^+, U_n^- \in \mathcal{U}_n^-,\ \forall n \in \mathcal{N}, \end{aligned} \tag{2.1}$$

where we assuming $h_n^{\leq}$, $h_n^{=}$, $g_e^{=}$ and f_n are affine functions. We therefore aim to design a Language L such that any problem instance that fits into the formulation above can be encoded in the language and vice-versa.

Chapter 3

Related Work

Chapter abstract: *In this chapter, we review the main approaches used in optimization modeling, discuss their advantages and drawbacks and compare them with the modeling language proposed in this work.*

The goal of this work is to design and implement a modeling language that natively handles the definition of problems involving the optimization of discrete-time dynamical systems over a finite time horizon and exhibiting a natural block structure that may be encoded by a sparse connected graph. Exploiting the latter is particularly useful, as it enables practitioners to build and store models in a modular and decentralized way. This also facilitates post-processing and result analysis.

We start by reviewing the most common approaches used in optimization modeling [11], and compare them with the proposed approach next.

First, in so-called unstructured approaches, the data, the model and/or the solver are all combined in one or several interacting pieces of code written in a programming language such as `C`, `C++` or `Fortran`. Such approaches are usually very low-level, in the sense that matrices defining the objective function and the constraints are directly built from data by the user in order to instantiate a problem. Custom code is then used to either solve the problem directly or convert the matrices into a format that is readable by an off-the-shelf solver. This approach lacks of readability and can be difficult to understand and maintain.

Second, in so-called table or spreadsheet approaches, the data is filled into the table and the mathematical structure is derived from it. This approach is the one preferred in the business world with tools such as Excel sheets. These usually do not scale-up very well and the class of problems, that can be tackled, is very narrow.

The third approach involves the use of the algebraic modeling languages (AMLs). They enable the encoding of optimization problems in a way that resembles the typical mathematical notation and are solver-independent. They typically work with an internal representation of the problem that can be converted to interface with a variety of off-the-shelf solvers. In AMLs,

we can distinguish two main approaches standalone AMLs or integrated AMLs that come in the form of packages available in scripting languages like Python and Julia (i.e. Pyomo [12] and JuMP [13]). This approach's main drawback is the fact it needs a little programming background. The standalone AMLs differ as they do not need any specific programming background. AMPL [14], GAMS [15] and AIMMS [16] are three of the best known standalone algrebraic modeling languages that deal with a wide range of optimization problems. In both approaches, the usage is quite simple and close enough to the equivalent mathematical formulation. However, they do not possess any structure in the problem encoding. Furthermore, in the standalone case, these AMLs are commercial software and their inner workings are opaque to the user.

In addition to these three approaches, it is worth mentioning the existence of object-oriented modeling languages (OOMLs). These differ from AMLs as they posses an object-oriented structure where one starts by defining a set of basic blocks that need to be assembled to create the full model. Hence, its formulation is not close to the mathematical one, but this approach has other advantages. For example, this paradigm makes it possible to construct models in a modular, decentralised and collaborative way. Such properties are particularly appealing to the task at hand. An example of OOML is the Extend tool [17].

We therefore propose the GBOML, which can be viewed as a stand-alone hybrid language including key features of both AMLs and OOMLs. GBOML exploits the strengths of both approaches. Indeed, it natively supports the definition of blocks that are assembled at a later stage, in a fashion similar to OOMLs. In addition, the formulation of each block can be defined using algebraic expressions and indexing features typically found in AMLs. The model formulated in GBOML is also solver-independent. Furthermore, it is open-source, it enables both researchers and practitioners to get a full insight on the inner workings of our solution.

Chapter 4

Theoretical Background

Chapter abstract: *In this chapter, we give the theoretical background necessary for understanding the implemented language and the way we solve problems in the language. We introduce concepts from the theory of computation and compilers.*

To fully understand the implemented solution and the novel modeling language, we need to lay some theoretical basis from the theory of computation and from compilers. We first need to understand some basic concepts as language, regular expression and grammar to be able to fully understand how a compiler works.

The theory of computation focuses on answering the question :

"what are the limitations of computer ?"

It is interested in the study of the class of problems that can be solved by a computer. One sub-branch of the theory of computation is formal language theory and the equivalent machines that can accept these type of languages. In the Section 4.1, we introduce all the concepts of regular and context-free languages necessary for the further understanding of the compiler. In Subsection 4.1.4, we introduce also the Extended Backus-Naur Form notation used for language definition.

The compiler's theory directly involved in the implemented solution. The compiler is just a particular program that takes advantage of the theory of computation concepts to translate a program from a source language to the equivalent program in a target language. In section 4.2, we introduce the basic ideas behind a compiler and its inner working.

4.1 Theory of Computation

In the following, we introduce concepts from the theory of computation. All the definitions found in the following were taken from P.Wolper's book on the theory of computation [18]. This section explains the basics units we will build on top of to go to more complex implementations.

First, we need to define the basic concepts. We define as symbol a single character or letter. It is the smallest building block considered. An alphabet Σ is a finite set of symbols. A word on an alphabet is a sequence of symbols belonging to the alphabet. The word of length 0 is called the empty word and is denoted by ϵ. A language L is a set of words defined over an alphabet Σ.

An intuitive example for the previous notions can be derived from the English language. A symbol would be any letter of the English alphabet. The English alphabet corresponds perfectly to our definition of an alphabet, namely a set of symbols. The word "abcdefghijklmnopqrstuvwxyz" is on the alphabet as all its symbols belong to the alphabet. However, it does not belong to the English language as it does not have any sense. We can define the words contained in any dictionary as the set of words belonging to the English language.

4.1.1 Regular Expressions and Language

In this section, we explain what regular expressions and languages are. A good understanding of these concepts is helpful to properly understand the compiler's lexer.

First we define some operators:

- the concatenation of two languages A and B, noted $A \cdot B$, is the language of the words containing all the words of language A concatenated to the words of language B, this can be seen as a Cartesian product. In other words, a word belongs to the language if its first part belongs to A and second part to B. For example, we consider $A = \{a, aa\}$ and $B = \{b, bb\}$, the concatenation of the two is given by, $A \cdot B = \{ab, abb, aab, aabb\}$,
- the Kleene star operation for languages, also noted *, is the concatenation of the set of words belonging to the language zero or multiple times. For the sake of illustration, we consider $A = \{aa, bb\}$. Then any combination of zero or multiple instances of words belonging to the language belong to $A^\star$, such as $\{\epsilon, aa, bb, aabb, bbaa, aaaa, bbbb, aaaaaa\} \subset A^\star$,
- the Kleene plus operation for languages, also noted $^+$, is the concatenation of the set of words belonging to the language one or multiple. If we take the previous example with $A = \{aa, bb\}$ then $\{aa, bb, aabb, bbaa, aaaa, bbbb, aaaaaa\} \subset A^+$,
- the union of two languages A and B, denoted $A \cup B$, is the language containing all the words of the language A and the language B. For example, if we consider $A = \{a\}$ and $B = \{b\}$, the union of A and B is given by $A \cup B = \{a, b\}$.

The set $\mathcal{R}$ of regular languages over an alphabet Σ is the smallest set of languages such that:

- $\emptyset \in \mathcal{R}$, and $\{\varepsilon\} \in \mathcal{R}$, where $\emptyset$ denotes the empty set and $\{\varepsilon\}$ is the singleton containing the empty word ε,
- $\{a\} \in \mathcal{R}$, $\forall a \in \Sigma$,
- let A,B $\in \mathcal{R}$, be two languages. Then, the union of the two $A \cup B \in \mathcal{R}$, the concatenation $A \cdot B \in \mathcal{R}$ and star concatenation $A^\star \in \mathcal{R}$
- regular languages can be represented by regular expressions. The regular expression can be viewed as a compact way of describing the words belonging to the language. To illustrate, we consider the language $L = \{ab, aab, aaab, aaaab, ...\}$ with all the words starting with one or several "a" symbols followed by a single "b", the regular expression equivalent is "a^+b".

In other words, a language is a regular language if it is the concatenation or union of two regular languages, if it is empty, if it contains the empty word only or one or several letters defined in the alphabet.

4.1.2 Finite Automaton

This section introduces the automaton and their link with regular expressions. We introduce the concepts of deterministic and non-deterministic finite automaton.

An automaton is a machine trying to replicate the execution of a program on a computer with its initial state, transition functions and accepting state.

A deterministic finite automaton (DFA) is made up of an input tape containing the word to check, a reading head that indicates the next symbol to be read, a set of states and a transition function to go from one state to another. The input tape is divided in blocks with each block containing a symbol. There is an initial state from where the automaton starts its execution, a set of intermediate states, a subset of which are accepting states. A transition function takes as input a state s_1 and a symbol a and returns the next corresponding state. If the automaton's execution ends in an accepting state, the word is accepted by the automaton.

Formally, a deterministic finite automaton (DFA) is defined by the 5-tuple $M = (Q, \Sigma, \delta, s_0, F)$ where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta : Q \times \Sigma \to Q$ is a transition function which taking a state in Q and letter in the alphabet Σ and returning a state in Q. In other words, from state $q_i \in Q$ when reading symbol $\sigma \in \Sigma$ we transition directly to state $q_j \in Q$ in a deterministic fashion,
- $s_0 \in Q$ denotes the initial state,

- $F \subset Q$ denotes the set of accepting states.

An expression is regular if and only if it is accepted by a DFA. This results in an equivalence between the two representations known as Kleene's theorem.

The execution of a deterministic finite automaton starts with the starting state s_0 and with the word that is considered in the input tape. The reading head reads the first character σ_0 on the tape and takes the corresponding transition $\delta(s_0, \sigma_0)$ to state $q_1 \in Q$. Then, it reads the next character σ_1 and arrives to the state determined by $\delta(q_1, \sigma_1)$ and so on until there is no more characters to read on the input tape. If the final state q_f is an accepting state $q_f \in F$, the word is accepted by the DFA otherwise it is rejected.

An example of a regular expression is $a^\star b$ introduced earlier, defined upon the alphabet $\Sigma = \{a, b\}$. The corresponding DFA is illustrated in Figure 4.1 with an example of execution.

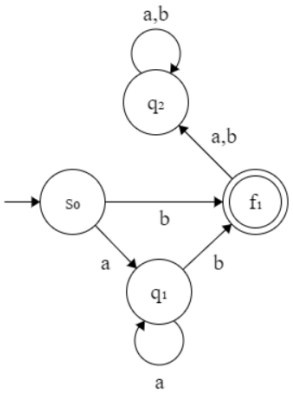

Figure 4.1: Automaton accepting the regular expression $a^\star b$ with s_0 being the starting state and f_1 the accepting state. To illustrate, we consider the execution to see if the word aab belongs to $a^\star b$. We start in s_0, we read "a" therefore we transition from (s_0, a) to q_1. Then we read another "a" on the input tape and transition from (q_1, a) to q_1. Finally, we read b, transition from (q_1, b) to the accepting state f_1. As there is nothing more to read and the final state is an accepting state, the automaton accepts the word aab as belonging to $a^\star b$. If another symbol (a or b) would have been read, as the word does not belong to the epression $a^\star b$, rather belongs to $a^\star b(a \cup b)^+$, we reach state q_2 from which no accepting state can be reached (bin state).

A non-deterministic finite automaton (NFA) is similar to a deterministic finite automaton except for two main differences. First, in opposition with DFA, a transition can be done by reading the empty word ϵ. Second, a transition can be done by reading several letters of the alphabet at once. Formally, the transition function is replaced by a transition relation denoted by $\delta \subset (Q \times \Sigma^\star \times Q)$, which implies that a for 3-tuple $(q_1, \sigma, q_2) \in \delta$, it is possible to go

from q_1 to q_2 by reading σ with $\times$ the Cartesian product. Note that every non-deterministic finite automaton can be turned into a deterministic one using the powerset construction [19]. The time complexity of the conversion from a NFA to an equivalent DFA is $O(2^m)$ where m denotes the number of states in the NFA. The main point of turning a NFA into a DFA is the computation time required to match a string of length $|w|$. In the deterministic case, as the transition function fully determines what is the next state to reach, the computation time to match a string of length $|w|$ is $O(|w|)$. However, for a NFA with $|Q|$ states, we don't have a transition function but only a transition relation, the non determinism leads to a time complexity of $O(|w||Q|^2)$ in the worst case.

These concepts will be useful to understand the compiler's lexer from a theoretical perspective, as well as its practical implementation.

4.1.3 Regular and Context-Free Grammar

In this section, we first introduce the concept of a grammar, underpins most of the work in this thesis.

A grammar is a 4-tuple $G = (V, \Sigma, R, S)$ where :

- V is the alphabet,
- $\Sigma \subseteq V$ is the set of terminal symbols with $V \setminus \Sigma$ being the set of non-terminal symbols,
- $R \subseteq (V^+ \times V^\star)$ is a finite set of production rules. A production rule consists of a symbol substitution. In other words, a production rule is written as $\alpha \to \beta$ where the symbols $\alpha \in V^+$ are replaced by the symbols $\beta \in V^\star$,
- $S \in V \setminus \Sigma$ is the start symbol.

In other words, V is the alphabet containing terminal and non-terminal symbols. A terminal symbol is one from which no production rule can be derived, written with a lowercase letter, and a non-terminal symbol, noted with uppercase letters, can be derived by a production rule. An example can be given by

- $V = \{S, a, A\}$
- $\Sigma = \{a\}$
- $R = \{S \to A, A \to aA, A \to \epsilon\}$
- S the start symbol

A word belongs to the language defined by a grammar if we can derive the word from its

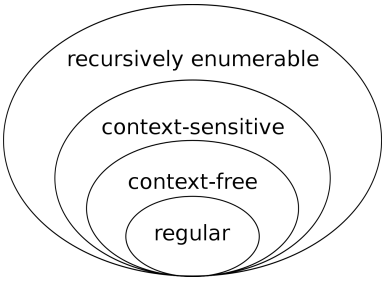

Figure 4.2: Chomsky hierarchy of the different types of grammars.
Image taken from : https://en.wikipedia.org/wiki/Chomsky_hierarchy

production rules. In the previous example, the derivation of $aaaa$ can be obtained as follows,

Symbols	Rule
S	$S \to A$
A	$A \to aA$
aA	$A \to aA$
aaA	$A \to aA$
$aaaA$	$A \to aA$
$aaaaA$	$A \to \epsilon$
$aaaa$	

There are several types of grammars according to what their right and left hand side hold in their transition. The most general notation for a transition is,

$$\alpha \to \beta$$

where there is no restriction on α and β. The regular grammars are those where α is a single non-terminal symbol and β is either a terminal symbol or a combination of a singular terminal symbol and a non-terminal one. In other words, of the form,

$$A \to wB$$
$$A \to w$$

with A,B $\in V - \Sigma$ and $w \in \Sigma^\star$.

The context-free grammars are those where α is a singular non terminal symbol and there is no restriction on β. Basically, any grammar which contains a production rule whose left-hand side is not a lone non terminal symbol is not context free. Chomsky hierarchy shows the set inclusions between the different types of grammars in Figure 4.2. As a language is regular if and only if it can be derived from a regular grammar and a language is context-free if and only it can be derived from a context-free grammar, being regular ensures that that language is context-free. However, if a language is context free, it is not necessarily regular. For example, $a^n b a^n$ is the language with the same number of 'a' symbols in the beginning and the end. $a^n b a^n$

is a context-free language but not regular as for the second part of the word, the next state does not only depend on the symbol on the tape and the current state but also the number of previously read 'a' symbols. This added form of counter memory can not be archived with regular languages.

4.1.4 BNF Form

The Backus–Naur form (BNF form) is a particular type of notation for context-free grammars. Instead of writing a rule as before

$$A \to \beta,$$

the rules are written as

$$< A > \quad ::= \beta$$

with A still being a non-terminal symbol and β being any combination terminal or non-terminal of symbols. In this notation, non-terminal symbols are put between '<.>' and if a non-terminal left-hand side has several associated rules, these are gathered in the right-hand side and separated by '|'.

If we consider the set of rules,

$$\begin{aligned} S &\to A \\ A &\to aA|B \\ B &\to b \end{aligned}$$

we would get,

$$\begin{aligned} &< S > ::=< A > \\ &< A > ::= a < A > | < B > \\ &< B > ::= b \end{aligned}$$

The extended BNF (EBNF) form is an extension where several notations are added or removed to ease the writing. In particular, we list the symbols used later in this manuscript

- '<.>' is not used anymore
- **,** is used for concatenation
- **{}** means that what is inside is considered 0 or multiple times
- **[]** means that the element inside can either be defined once or ignored
- **""** the element inside is a terminal string

In the extended BNF form, the example is given by,

```
S = {"a"}, "b";
```

4.1.5 Pushdown Automaton

The pushdown automaton section gives a better understanding of the difference between context-free and regular expressions. The compiler's parser implements a kind of pushdown automaton, hence, the importance of the following concepts.

A pushdown automaton is similar to a non-deterministic finite automaton with an additional stack on which symbols can be pushed. It means that an additional form of memory is added to the previous deterministic finite automaton in which transitions depended only on states and read symbols. Formally, the pushdown automaton is defined by the seven-tuple $M = (Q, \Sigma, \Gamma, \Delta, Z, s, F)$, where

- Q the finite set of states,
- Σ the input alphabet,
- Γ the stack alphabet,
- $Z \in \Gamma$ the initial symbol on the stack,
- $s \in Q$ the initial state,
- $F \subseteq Q$ the accepting states
- $\Delta \subset ((Q \times \Sigma^\star \times \Gamma*) \times (Q \times \Gamma*))$ are the finite set of transition relations. Intuitively, from each state $q_i \in Q$, by reading one or more symbols $\sigma \in \Sigma$ on the tape, by reading one or more symbols on the stack $\gamma_i \in \Gamma$ we can access state $q_j \in Q$ by writing $\gamma_j \in \Gamma$ on the stack.

A grammar is context-free if and only if it is accepted by a pushdown automaton.
To illustrate this, we consider the following set of rules,

$$S \to A$$
$$A \to b$$
$$A \to aAa$$

that defines the context-free language $a^n b a^n$. An example of a pushdown automaton execution to check if aabaa belongs to the language is given below,

Stack	*Tape*	*Rule*	*Action*
S	$aabaa$	$S \to A$	$Push(A)$
SA	$aabaa$	$A \to aAa$	$Read(a), Push(A)$
A	$abaa$	$A \to aAa$	$Read(a), Push(A)$
AA	baa	$A \to b$	$Read(b)$
AA	aa	$A \to aA$	$Pop(A)$
A	a	$A \to aAa$	$Pop(A)$
S			Accepted

The equivalent automaton is illustrated in Figure 4.3.

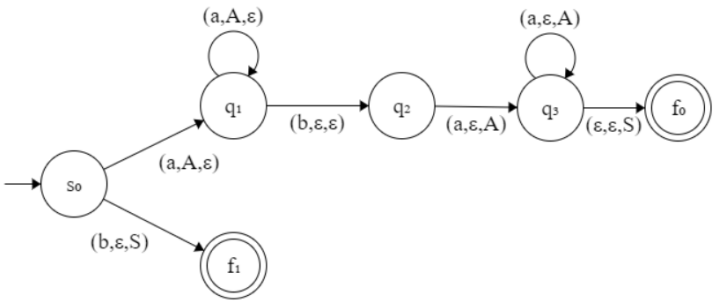

Figure 4.3: Pushdown automaton accepting the language $a^n b a^n$, the initial state is denoted by s_0, the accepting states are f_0, f_1 and the transitions are made of the 3-tuple (a_i, w_i, r_i) with a_i being the letter read from the input tape, w_i the element pushed on the stack and r_i the element popped from the stack.

4.2 Compiler

In this section, we define the concept of compilers and explain its components. The definitions contained in the following section are those of the compiler course given at the University of Liege by Pr P.Fontaine [20] and from the "Compilers : Principles, Techniques, & Tools" book [21].

First, a compiler can be defined as a program that takes as input a source program written in a given language L_s and translates it into an equivalent program written in a second target language L_o. It also checks mistakes in the source program. The compiler is typically made of five parts:

- **The lexer** takes as input the source program as a stream of characters and groups the character in meaningful sequences. To each sequence, it associates some more information as a type, a line at which it was defined, and turns them into an object called token. It outputs a stream of tokens
- **The parser** takes as input the stream of tokens and builds an intermediate tree-like representation based on the grammatical structure of the tokens.
- **The semantic analyzer** checks the semantic errors that can occur in the previously created intermediate structure.
- **The intermediate representation** augments the previous intermediate structure or/and adds more intermediate representation of the program.
- **The code generation** converts all the gathered information and intermediate representation into a program in the target language.

4.2.1 Lexer

As previously explained, the lexer takes as input a source program as a stream of characters and groups these into meaningful sequences also called lexemes. Each lexeme is labeled with a token, a tuple made of an identifier and some addition information. Each token is defined by a regular expression. If a lexeme satisfies the regular expression of a given token, it will be labelled with the corresponding token. If an unknown character or sequence of characters is encountered, the lexical analyzer emits an error. In a more intuitive approach, the lexer cuts a stream of characters into meaningful words.
To do so, the lexical analyzer builds a DFA for each regular expression. It then links all these DFA into one big non-deterministic finite automaton by adding a new start state and putting ϵ transitions from this new start state to the start state of each DFA. The non-determinism introduced by the ϵ transitions is eliminated by converting the NFA to a DFA. We feed into this new DFA's input tape the stream of characters and it will do the corresponding transitions typically, following a longest match rule, meaning that it will keep on reading the characters until it is stuck and go back to the last accepting state visited to determine the token type.

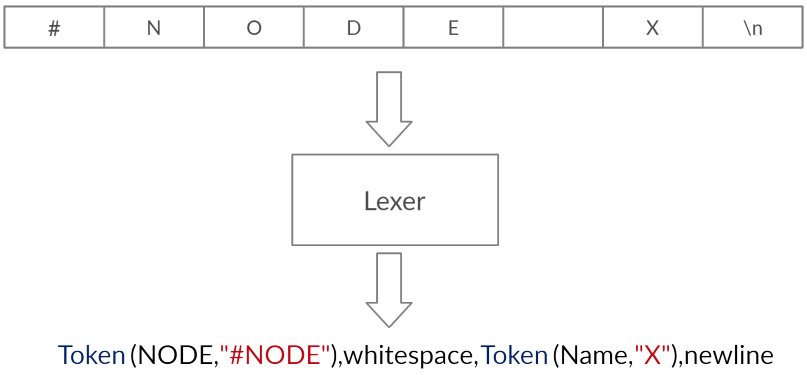

Figure 4.4: Example of the lexer for the GSML language introduced section XX taking a stream of characters as input and produces the corresponding output. The NODE token is used to introduce the definition of a node object. A whitespace is used for counting number of characters read but is ignored otherwise. The token NAME corresponds to the definition of a node name. Newline does not return a token but is used to keep track of the line number.

4.2.2 Parser

The parser receives as input a stream of tokens and outputs an intermediate representation, also called an abstract syntax tree, based on a grammar. In a more intuitive way, this step corresponds to building sentences that make sense out of words. The grammar of the parser must be context free and unambiguous to enable the usage of efficient parsing algorithms. Without these algorithms and assumptions, the time complexity to parse a string of n terminals is $O(n^3)$. Fortunately, as most programming languages make a single left-to-right scan over the input, only one look ahead at a time is enough for constructing the pieces of the parse tree by using a kind of pushdown automaton. There are two ways of building the syntax tree,

- **Top-down parsing:** build the tree elements from the root and go to the leafs
- **Bottom-up parsing:** build the tree elements from the leafs up to the root

Even though the top-down parsers are preferred when building a parser by hand, most software tools are built using bottom-up parsers as they offer more options when it comes to translation schemes and they handle a larger class of grammars.

4.2.3 Semantic Analysis

The semantic analyzer checks the intermediate representation for several types of mistakes linked to the context. If we continue the more intuitive parallel, this step would correspond to checking if the sentences make sense between themselves. As the previous step only deals with context-free grammars, the semantic analyzer checks the semantic consistency of the input. Furthermore, it augments the intermediate representation by adding more information.

4.2.4 Intermediate Representation

The intermediate representation is a step intertwined with the previous one as the syntax tree is already an intermediate representation of the input program. In this step, several other representations can be processed. Typically, we translate all the gathered information to a lower level representation easily convertible to the target language.

4.2.5 Code Generation

Finally, the last step is converting all the gathered information and the previous representation in a program written in the target language. Several steps of code optimization can also be done to optimize the output program.

Chapter 5

A Graph-Based Optimization Modeling Language

Chapter abstract: *This chapter contains an explanation of the implemented language that tackles our problem definition. The proposed language is called Graph-Based Optimization Modeling Language (GBOML)*

Recall that the purpose of this work is to design and implement a language to represent a certain class of linear optimization problems, namely, problems possessing a block structure that can be encoded by a sparse connected graph. To answer this problem statement defined in Chapter 2. We implement a hybrid language somewhere in between an algebraic modeling language and an object-oriented modeling language, called Graph-Based Optimization Modeling Language or GBOML. GBOML provides an easy and understandable framework for formulating the aforementioned mentioned class of problems. In Section 5.1, we formally present the language and its grammar. Section 5.2 contains a tutorial-like explanation of the language semantics.

5.1 Language Definition

This section contains a definition of the Graph-Based Optimization Modeling Language. First, we define the basic units, or blocks that we use to ease the writing of the different elements of the language. Second, from these atomic blocks, we define the list of tokens. Third, from these tokens, we build the grammar of our language. Chapter 4 provides the necessary background for fully understanding the following section.

Basic Units

The basic blocks, expressed in the EBNF notation introduced in Section 4.1.4, are defined as follows:

```
lowercase = "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"
            |"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"x"|"y"|"z";
uppercase = "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"
            |"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"X"|"Y"|"Z";
digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
dollar = "$"
letter = lowercase | uppercase | "_";
```

We define the blocks composed of space (`sp`), horizontal tabulation (`tab`), line feed (`lf`), form feed (`ff`) and carriage return (`cr`) by the name of whitespace:

```
whitespace = { sp | tab | lf | ff | cr };
```

Tokens

GBOML accepts one type of string identifiers, used for variables, parameters and node names.

```
id = letter,{letter,"_",digit,dollar};
```

The filenames are defined using quotation marks.

```
quote = "\""; //ASCII character 42
character = letter | digit | "." | "/";
filename = quote, character,{character},quote;
```

Numbers can be either be represented as floats or as integers, with the corresponding rules given below.

```
integer = digit,{digit};
float = integer,".",{integer};
```

The following strings are defined as keywords of the language, that is, words that are reserved and can only be used in a special context. These words, when encountered, return a special reserved token:

```
min = "min";            node = "#NODE";             step = "step";
max = "max";            param = "#PARAMETERS";      for = "for";
input = "input";        cons = "#CONSTRAINTS";      and = "and";
```

```
output = "output";      var = "#VARIABLES";          mod = "mod";
internal = "internal";  obj = "#OBJECTIVES";         where = "where";
in = "in";              time = "#TIMEHORIZON";       not = "not";
horizon = "horizon";    links = "#LINKS";            or = "or";
import = "import";
```

The language also supports a set of special symbols, mostly mathematical operators. These are given as follows with their respective tokens:

```
plus = "+";     minus = "-";       lpar = "(";       semicolon = ";";
mult = "*";     divide = "/";      rpar = ")";       colon = ":";
pow = "**";     equal = "=";       lbrac = "[";      deq = "==";
comma = ",";    dot = ".";         rbrac = "]";      neq = "!=";
lcbrac = "{";   rcbrac = "}";      leq = "<=";       low = "<";
geq = ">=";     greater = ">";
```

Any whitespace is ignored, meaning that no token is associated to them. A comment is defined as a sequence of tokens introduced by "//" and ending with a line feed.

Grammar

The previously-defined tokens must respect a set of rules called the grammar. The concept of grammar is explained in Section 4.1.3. The full grammar can be expressed as:

```
start = [time],{node},[links];

time = "#TIMEHORIZON", id, "=", expression;

node = "#NODE", id, [def_parameters], def_variables,
       [def_constraints],[def_objectives];

def_parameters = "#PARAMETERS", {parameter};
parameter = id, "=", expression;
          |id, "=", "{", term, {",", term},"}";
          |id, "=", "import", filename;

def_variables = "#VARIABLES", variable, {variable};
variable = "internal", ":", identifier_obj;
         |"input", ":", identifier_obj;
         |"output", ":", identifier_obj;

def_constraints = "#CONSTRAINTS", {constraint, ";"},
                  constraint, [";"];
constraint = expression, "=", expression, [for_loop], [condition];
           |expression, "<=", expression, [for_loop], [condition];
           |expression, ">=", expression, [for_loop], [condition];

for_loop = "for", id, "in", "[", expression, ":", [expression, ":"],
          expression, "]";
condition = "where", bool_condition;
```

```
def_objectives = "#OBJECTIVES", {objective};
objective = "min", ":", expression;
          |"max", ":", expression;

links = "#LINKS", {link};
link = id, "=", id, {",", id};
     |id, ".", id, "=", id, ".", id, {",", id, ".", id};

bool_condition = bool_condition, "and", bool_condition;
               |bool_condition, "or", bool_condition;
               |"not", bool_condition;
               |expression, "!=", expression;
               |expression, "<", expression;
               |expression, ">", expression;
               |expression, "<=", expression;
               |expression, ">=", expression;
               |expression, "==", expression;

expression = "(", expression, ")";
           |expression, "+", expression;
           |expression, "-", expression;
           |expression, "*", expression;
           |expression, "/", expression;
           |expression, "**", expression;
           |mod, "(", expression, ",", expression, ")";
           |"-", expression;
           |term;

term = identifier_obj;
     |integer;
     |float;

identifier_obj = id ;
               |id, "[", expression, "]";

integer = digit, {digit};

float = digit, ".", {digit};

id = letter,{letter,"_",digit,dollar};
```

For the sake of clarity, some tokens were not used directly in the grammar definition. Instead, a string was used, e.g. using the string "=" instead of the equal token, as it gives better insight into in each definition.

5.2 Tutorial and Semantics

This section contains a tutorial on the language semantics that provides further insight into the defined language and its usage.

The language in itself is composed of three distinct blocks:

```
start = [time],{node},[links];
```

Each block corresponds to a particular part of the formalism defined in Chapter 2. The `[time]` rule is used to define the horizon, the `{node}` rule defines the set of nodes of the graph and the `[links]` rule defines the set of edges between the graph nodes.

Time

The time horizon definition is not mandatory. If defined, this block should be the first one in the document. If not defined, the time horizon is set to one by default. The time horizon is defined in the grammar by the rule:

```
time = "#TIMEHORIZON", id, "=", expression;
```

The only identifier allowed in the time horizon definition is `"T"`. The right-hand side expression should evaluate to a positive integer. Entering a positive float results in the value being rounded to the nearest integer and a warning is raised. No negative value is allowed. To illustrate this, a horizon of 24 time steps can be declared as:

```
#TIMEHORIZON          #TIMEHORIZON          #TIMEHORIZON          #
    TIMEHORIZON
T = 24                T = 12*2              T = -40/2+44          T = 24.01
```

They all lead to the same horizon with the last one raising a warning. The horizon is divided in `T` timesteps with the first timestep being 0. The identifier `t` is also automatically defined and corresponds to each time step in the horizon, $\texttt{t} \in \{0, ..., \texttt{T} - 1\}$.

Node

A node $n \in \mathcal{N}$ is defined in the GBOML grammar through four fields, namely, `#PARAMETERS`, `#VARIABLES`, `#CONSTRAINTS` and `#OBJECTIVES` and an identifier. The fields `#PARAMETERS`, `#CONSTRAINTS` and `#OBJECTIVES` are not mandatory for each node. Each node must have a unique identifier. In addition, there must be at least one variable in each node. The typical node structure, for node $n \in \mathcal{N}$, is given by:

```
1 ...
2 #NODE n
3 #PARAMETERS
4 ...
5 #VARIABLES
6 ...
```

```
#CONSTRAINTS
...
#OBJECTIVES
...
```

Parameters

A parameter $p \in \mathcal{P}$, where $\mathcal{P}$ is the set of parameters, can be defined as:

```
...
#PARAMETERS
p = ...
...
```

The scope of a parameter is the node in which it is defined. Accessing a parameter from another node is therefore not possible. There are two main types of parameters, either scalar parameters or vector parameters that are indexed (that we also call indexable parameters in the following). A scalar parameter is defined by the rule :

```
parameter = id, "=", expression;
```

The expression is evaluated and its value is assigned to the identifier in the definition. The expression can contain other previously-defined parameters as they are evaluated in their order of appearance. Thus, using a parameter p_1 in the definition of p_2 is only valid if p_1 has been previously defined for this node, for example:

```
...
#PARAMETERS
p_1 = 12
p_2 = p_1*10/32
...
```

The indexable parameters can either be defined by importing a file or by directly defining a set of values, respectively:

```
parameter = id, "=", import, filename;
parameter = id, "=", "{", term, {",", term}, "}";
```

The file designated by the `"filename"` must exist and must only contain numbers separated either by a semicolon, space or linefeed. The values in the file are read from left to right and from top to bottom. When using the second rule, the terms inside the definition can reference previously-defined parameters. With the previously-defined parameters `p_1` and `p_2` one could define a new parameter `p_3` as:

```
...
p_3 = {p_1,p_2,45}
...
```

We call these parameters indexable, since their identifiers must be followed by an index value in brackets `[.]` to access one of their entries. Note that the index returning the first entry is

0 and not 1. In the previous example, the parameter `p_3` is a three-value vector. The value at the first index would be equal to `p_1`, the value at the second index to `p_2` and the third to 45. In other words:

$$\texttt{p\_3}[0] = \texttt{p\_1} \qquad \texttt{p\_3}[1] = \texttt{p\_2} \qquad \texttt{p\_3}[2] = 45.$$

The indexed values can be used for defining another parameter. However, using `p_3` directly to define another parameter is not allowed. For example, in order to define a parameter `p_4`, one cannot write:

```
...
p_4 = 5*p_3
...
```

This limitation is rather a choice as operations for vectors have not been considered.

Variables

Input, internal and output variables denoted by $u \in \mathcal{U}_n^+$, $x \in \mathcal{X}_n$ and $w \in \mathcal{U}_n^-$, respectively, can be defined as:

```
def_variables = "#VARIABLES", variable, {variables};
variable = "internal", ":", identifier_obj;
         |"input", ":", identifier_obj;
         |"output", ":", identifier_obj;
```

The variable type is given by the first token in the definition. The variables are defined as vectors with T entries, therefore, they are also indexable. An example of definition is:

```
#VARIABLES
input : u
internal : x[t]
output : w
...
```

where `u` is an input variable, `x[t]` an internal variable and `w` an output variable. Writing either `x[t]` or just `x` leads to the exact same definition of a variable `x` with `T` entries. Only the index `t` is permitted for variable definitions.

Constraints

Constraints are defined by the following rules:

```
def_constraints = "#CONSTRAINTS", {constraint, ";"},
                  constraint, [";"];
constraint = expression, "=", expression, [for_loop], [condition];
           |expression, "<=", expression, [for_loop], [condition];
           |expression, ">=", expression, [for_loop], [condition];
```

The basic unit of a constraint is made up of the 3-tuple `expression - sign - expression`. The general linear constraint

$$a_1x_1 + a_2x_2 + ... + a_nx_n = b$$

can be written as:

```
#VARIABLES
internal : x
...
#CONSTRAINTS
a_1*x[1]+a_2*x[2]+...+a_n*x[n] = b;
```

If $n \geq T$ or $n < 0$, the constraint is simply ignored. A constraint must contain at least one variable.

When using a variable `x` in a constraint, if the index for which the variable is considered is not mentioned, we consider it for all the timesteps and it is thus equivalent to writing `x[t]`. Therefore, the two constraints:

```
#VARIABLES
internal : x
...
#CONSTRAINTS
x = b;
x[t]=b;
```

are equivalent and lead to,

$$\begin{aligned} x[0] &= b \\ x[1] &= b \\ &\vdots \\ x[T-1] &= b \end{aligned}$$

The same is true for indexable parameters. Indeed, if no index is mentioned, they are considered for every index in the horizon. Therefore, if the parameter does not contain `T` values, an error is raised. Moreover, the index employed with variables and vector parameters must be a positive integer, otherwise an error is raised. It shall be noted that a constraint for which one of the indexes is out of bounds is ignored. In other words, if there exists a value `t` in the interval `[0,T-1]` such that one variable in the constraint tries to access a value outside the `[0,T-1]` interval that constraint is ignored for that particular `t`. For example if one writes, for $T = 3$,

$$x[t+1] + y[t] = 1$$

it defines the constraints

$$x[1] + y[0] = 1$$
$$x[2] + y[1] = 1$$
$$x[3] + y[2] = 1$$

The last constraint is ignored as one of the indexes is out of bounds. Hence, caution must be observed when defining constraints in order to get the right indices. For example, we consider the two constraints

$$x[t-1] + x[t] = 1$$
$$x[t] + x[t+1] = 1$$

From a mathematical standpoint, if we pose $t' = t + 1$, we can convert the first constraint into the second one. However, in our framework, the first one is not evaluated for the timestep `t = 0`, whereas, the second one is not valid for `t=T-1`. Therefore, these constraints are not equivalent.

Constraints can be defined for a subset of time steps by using the rule:

```
for_loop = "for", id, "in", "[", expression, ":", [expression, ":"],
           expression, "]";
```

This rule can either contain two or three expressions. If it contains two expression, the in-between square bracket expression, `[expression, ":"]` is ignored, the first expression corresponds to the beginning of the interval and the second one to the end. If we have three expressions, each respectively correspond to the beginning, the chosen step and the end of the considered interval. These two/three values must be positive integers with the beginning lower than the end and the end lower than the horizon. If the end value is bigger than the horizon, it is automatically put back to horizon value and a warning is raised.

Another way to express a constraint in an interval is using a condition given by the rule:

```
condition = "where", bool_condition;
```

The constraint is evaluated for any value `t` in the interval that satisfies the condition in `bool_condition`. The `bool_condition` can regroup several boolean predicates by using `OR` and `AND`. `NOT` can be used to negate predicates.

All variables are defined as vectors where each index is associated to one time step of the optimization horizon. Having a variable constant over the time horizon can be achieved by adding an additional equality constraint on the variables. For example, assuming we want a constant internal variable x, we can write:

```
1 #VARIABLES
2 internal : x
3 ...
4 #CONSTRAINTS
5 x[t]=x[0];
```

Finally, note that, all the constraints must be separated by semicolons with the last constraint not necessarily needing it.

Objectives

The objective function, defined as f_n in Equation 2.1, can be maximized or minimized, which is expressed as follows:

```
objective = "min", ":", expression;
          |"max", ":", expression;
```

The `expression` must be linear with respect to the variables and is summed for every `t` in the horizon. Therefore, writing:

```
#VARIABLES
internal : x
...
#OBJECTIVES
min : x
```

is equivalent to

$$\min f_n = \sum_{t=0}^{T-1} x[t]$$

There must be at least one valid constraint and one valid objective in the whole program. All the objectives from each node are gathered and summed to form the global objective, corresponding to the total summation of Equation (2.1).

Links

Links are defined by the rules:

```
links = "#LINKS", {link};
link = id, "=", id, {",", id};
      |id, ".", id, "=", id, ".", id, {",", id, ".", id};
```

In the simplest case, we have:

```
link = id, ".", id, "=", id, ".", id;
```

where the first identifier is a node name, the second an output variable from the previous node, the third one a second node name and the last one an input variable. Put simply, we have `Node1.output = Node2.input`. Linking two variables adds equality constraints between the two for the whole horizon. It is not mandatory for every `input/output` variable to be linked. Linking several inputs to one output is also possible. This is achieved by adding multiple input variables on the right-hand side of the equality in the link definition.

The second rule:

```
link = id, "=", id, {",", id};
```

is a simplification that can be used to link two nodes. If `Node1` possesses only one output variable and `Node2` only one input variable, we can directly write `Node1 = Node2`.

5.3 Making the Grammar Unambiguous

We make the grammar unambiguous by defining precedence rules. Basically, two rules are ambiguous if there is no way to know which rule shall be applied first over which one. In our particular case, these concern operators in expressions or boolean conditions.

Operator	Priority	associativity
`**`	1	left
`unary minus`	2	right
`*` `/`	3	left
`+` `-`	4	left

`=`
`<=`
`>=`
`<`
`>`
`!=`
`==`
	5	non-associative

`NOT`	6	right
`AND`	7	left
`OR`	8	left

Table 5.1: Precedence rules of the language based on priority and associativity

Therefore, we define the priority and associativity of our operators. The priority determines the order of evaluation for different operators. Whereas, the associativity determines the order of evaluation for similar priority operators. To illustrate, we consider the example where we have `expression1 op1 expression2 op2 expression3`. If operator `op1` has a higher priority than operator `op2`, we first perform `expression1 op1 expression2`. If these two operators have the same priority, their order of evaluation is determined by their associativity. Two operators with the same priority also have the same associativity. If the operator `op1` is left associative, the expression derived from `expression1 op1 expression2` is built first and if it is right associative, it is the expression derived from `expression2 op2 expression3`. If one operator is non-associative, that means that writing `expression1 op expression2 op expression3` raises an error. The precedence rules defining priority and associativity are given in Table 5.1. An example of a unary minus is : a = -b where - is the sign of b. Theses precedence rules correspond to the mathematical precedence rules. It should be noted that the using of `(.)` can change the order of priority of the rules.

Chapter 6

Implementing the Language

Chapter abstract: *In this chapter, we detail the choices of implementation that were made to be able solve an optimization problem expressed in the GBOML grammar. First, we motivate the choices of the GBOML synthax. Second, we explain the internal representation of the problem statement given in Equation* (2.1)*. Third, we explain how GBOML input is converted into the internal representation, through usage of a compiler. Finally, we implement a link between the internal representation of the problem statement and of the standard form used for solving linear problems with external solvers.*

The previous chapter presented the language implemented to enable the encoding of our class of problems. We discuss the language and present a way to solve problems written in the language. Therefore, we propose a multi-step solution composed of

- a file written in the Graph Based Optimization Modeling **Language**
- a **compiler** for the conversion of the above mentioned language into a matrix form equivalent
- a **communication interface** to link the matrix form to a solver in order to solve it and retrieve the corresponding results

The equivalent is shown in Figure 6.1 with the communication interface sending and retrieving the solution of a given problem from the solver.

First, we explain the language choices by going through the needs and evolution to the current solution. Second, we explain how the language is represented in computer memory formally. Third, we go through the translation from an input file written in GBOML to the internal representation. We end by proposing a way to retrieve a solution from the internal representation.

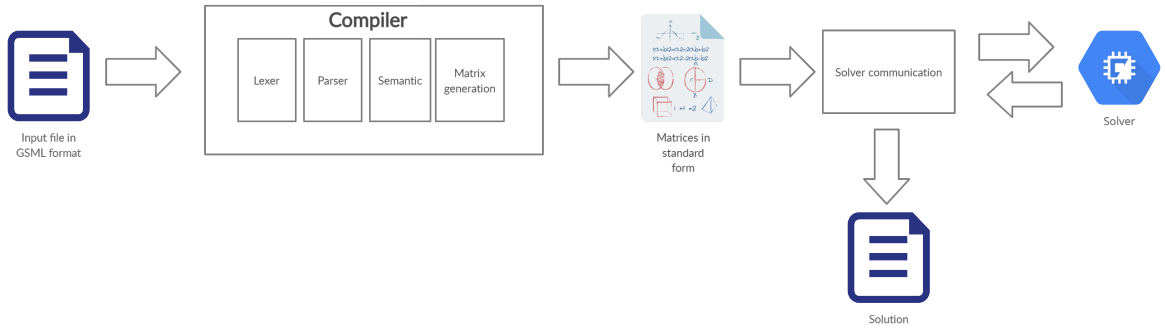

Figure 6.1: Diagram of the implemented solution. The initial input is a file written in human-readable format. It is sent to a compiler that converts it in the mathematical formalization. The equivalent matrices are send to a solver via a communication link that retrieves the solution.

6.1 Language Choices

In this section, we explain how the language evolved and what were the functionalities required and problems encountered that led us to the actual language. We go through the basic structure, the necessity of implementing indexable parameters, the constraints separation, the necessity of a more precise horizon and the syntax for links. By discussing these, we already give a good overview of most problems encountered when developing the language.

Structure

First, we wanted a simple grammar with a graph based implementation. To keep things as simple as possible, the keywords : `"#TIMEHORIZON"`, `"#NODE"`, `"#PARAMETERS"`, `"#VARIABLES"`, `"#CONSTRAINTS"` and `"#OBJECTIVES"` were first introduced. These gave the basic structure:

```
#TIMEHORIZON
...
#NODE ...
#PARAMETERS
...
#VARIABLES
...
#CONSTRAINTS
...
#OBJECTIVES
...
```

In the first place, links between variables were done implicitly. One could link two variables in different nodes by just using the same name. Typically, as shown in the example below, there would be an automatic connection between A.x and B.x:

```
#NODE A
...
```

```
#VARIABLES
input : x
...
#NODE B
...
#VARIABLES
output:x
...
```

However, this way of proceeding was too restrictive in many aspects. First, for example in system with multiple nodes producing a good such as electricity, one would rather model the links by himself to better define the graph structure. Second, users could find such an implicit adding of constraints as unnatural or counter intuitive. Therefore, the keyword `"#LINKS"` was added with link definition being explicit. With the structure now being:

```
#TIMEHORIZON
...
#NODE ...
#PARAMETERS
...
#VARIABLES
...
#CONSTRAINTS
...
#OBJECTIVES
...
#LINKS
....
```

Default Horizon

By not using the `#TIMEHORIZON` definition, we automatically put the horizon equal to 1. We decided this to enable users not interested by this addition horizon to still use our framework without having to worry about that additional expansion.

Parameters

For parameters, in the very beginning, only a single scalar value could be associated to a parameter. However in many applications, the value of the parameter may also be time dependent. In order not to have to encode T variables and T constraints for each time, we introduced vector parameters. For example:

```
#TIMEHORIZON T = 3
...
#NODE A
#PARAMETERS
a1 = 1
a2 = 2
```

```
a3 = 3
...
#VARIABLES
internal : x
...
#CONSTRAINTS
x[0] = a1;
x[1] = a2;
x[2] = a3;
...
```

With indexable parameters one can simply write:

```
#TIMEHORIZON T = 3
...
#NODE A
#PARAMETERS
a = {1,2,3}
...
#VARIABLES
internal : x
...
#CONSTRAINTS
x[t] = a[t];
...
```

For a very large time horizon, writing by hand each value in an indexable parameter can make the file less readable. Furthermore, some data are available to the modeler as time series in already existing external files. Therefore, the import option was added to directly import a parameter's values from a file. An example is given in the following:

```
...
#NODE A
#PARAMETERS
a = import "file.csv"
...
```

Modulo

In order to deal with the particular case where a profile is repeated several times in the time horizon, we introduced the modulo operator. To grasp the idea behind it, we consider the following example, we can consider a student that wants to optimize his number of sport hours by still keeping 9 hours of sleep and working 7 to 8 hours a day with other precise constraints as the hours where he should eat or see friends or rest during a full week. As our student wants to sleep every day from 10PM to 7AM, we could have a sleeping profile given by:

```
#TIMEHORIZON T = 24*7
...
#NODE A
```

```
#PARAMETERS
sleep_hours = {1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1}
...
#VARIABLES
internal : sleeping
...
#CONSTRAINTS
sleeping[t] = sleep_hours[mod(t,24)];
...
```

We denote by ones the hours when the student is sleeping and 0 when he is awake. The variable sleeping is illustrated in Figure 6.2.

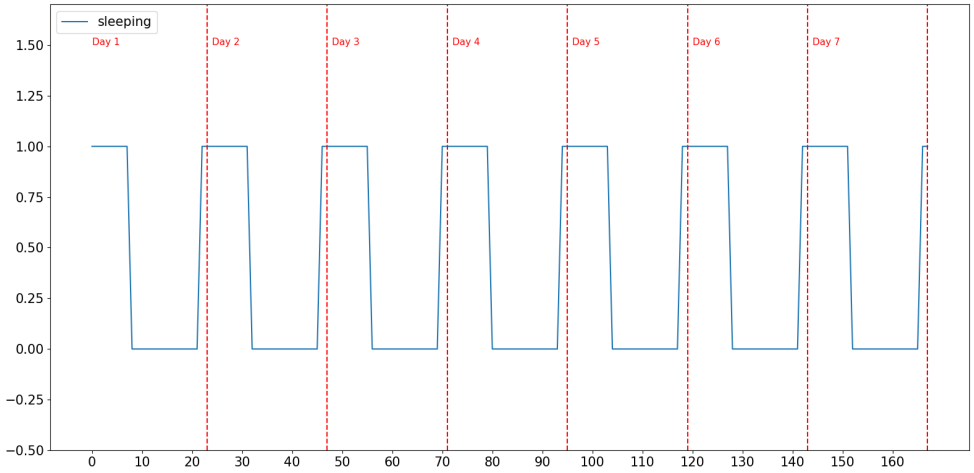

Figure 6.2: Sleeping variable for 7 days considering a profile where a student sleeps from 10PM to 7AM with 1 denoting that he is sleeping and 0 that he is awake.

The modulo operator resulted from this need of enabling cyclic repetitions inside the horizon. The previous implementation would have required to hand-write 168 constraints just for this particular case or to import a file with 168 values. These repetitions can be found in various cases from PV panels production to hormonal cycles.

Semicolon

The need of the semicolon at the end of a constraint was felt when the constraints were generalized from their initial form:

```
constraint = identifier_object "=", expression;
```

to enable a more general writing of constraints such as the standard form:

$$a_1x_1 + a_2x_2 + ... + a_nx_n = b$$

Hence, the more general constraint definition was introduced:

```
constraint = expression, "=", expression;
```

This definition comes with a minor issue, in the very particular case where the left-hand side expression starts with a unary minus. For example:

```
...
#CONSTRAINTS
u = y
-x-3=4*z
...
```

There is an ambiguity, as the linefeeds are not used as tokens, between these two constraints:

$$u = y \qquad\qquad u = y - x$$
$$-x - 3 = 4 * z \qquad\qquad -3 = 4 * z$$

Consequently, a semicolon was added in the end of a constraint , even tough it is an additional restriction in writing a problem. Another solution would have been to take linefeed as a token but it would also add constraints on the writing. By taking into account linefeed, one could not write a very long constraint on several lines as:

$$\begin{aligned}\texttt{expression} &= \texttt{expression} \\ &+ \texttt{expression} \\ &+ \texttt{expression}\end{aligned}$$

which is a restriction too.

Constraints Definition Over Time-Intervals

Every constraint is defined for the full interval `[0,T-1]`. For example:

```
#TIMEHORIZON
T = 1000
...
#CONSTRAINTS
u[t] = t;
...
```

corresponds to the following constraints:

$$\begin{aligned} u[0] &= 0 \\ u[1] &= 1 \\ &\vdots \\ u[T-1] &= (T-1) \end{aligned}$$

One can think of several problems where a subset of the interval `[0,T-1]` is needed. In that case, one can write each constraint individually. For example, lets consider a time horizon of 1000 and that a constraint needs to be defined for the 100 first constraints, one would have to write:

```
#TIMEHORIZON
T = 1000
...
#CONSTRAINTS
u[0] = y;
u[1] = y;
u[2] = y;
...
u[99] = y;
...
```

which can very quickly become too much work on a user for a big sub interval in the [0,T-1] interval. Therefore, a way of specifying these intervals was needed. The first considered solution was to enable a `Python` like implementation where the interval can directly be specified in the index of a variable. Such as:

```
...
#CONSTRAINTS
u[0:99]=y;
...
```

Although elegant, there are some particular cases where there could be some form of ambiguity. One could think of the following problem, we want to constraint the values x[0:9] to be equal to y[10:19]. This does not pose any problem as we can take easily associate x[0] with y[10], x[1] with y[11] and so on. However, there is a particular case where a problem occurs: when the predefined timestep value `t = 0, ..., T-1` is used as a coefficient. For example,

$$x[0:9] - y[10:19] = t$$

There is no clear way of knowing what is the value of `t` to be considered. This led us to the implemented solution where we added a keyword `"FOR"` to remove the ambiguity:

```
...
#CONSTRAINTS
x[t]-y[t+10]=t for t in [0:9];
...
```

Similarly, the `"WHERE"` condition was implemented to enable the consideration of discontinuous intervals and more advanced operations on `t`. The previous also be written as:

```
...
#CONSTRAINTS
x[t]-y[t+10]=t where t<10;
...
```

In this solution, the condition is evaluated for every `t` in the full interval, or the one specified, and the constraint is only considered for the values of `t` that satisfy the condition.

Output/Input Links

The link definition as `OUTPUT = INPUT` came from the fact that we wanted to link several variables directly though one equality such as:

```
...
#LINKS
Node1.variable = Node2.variable,Node3.variable
...
```

Therefore, we had to choose how we could enable to encode these type of general links. From a physical standpoint, linking an input to several outputs make little to no sense. Hence, the choice of writing links as `"OUTPUT = list of INPUT"` was made. However, as it is quite restrictive, this choice may be revised in the future to enable a more general writing of links.

6.2 Mathematical Representation

A representation for storing in computer memory the problem definition from Chapter 2 is required if we want to solve an optimization problem expressed in the GBOML.

As a reminder we have an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and an optimization horizon T with $\mathcal{N} \in \mathbb{N}$ being the node set and $\mathcal{E}$ being the edge set. Each node $n \in \mathcal{N}$ contains a set of input, internal and output variables respectively noted $\mathcal{U}_n^+ \subseteq \mathbb{R}^{d_n^+ \times T}$, $\mathcal{X}_n \subseteq \mathbb{R}^{d_n \times T}$ and $\mathcal{U}_n^- \subseteq \mathbb{R}^{d_n^- \times T}$, where d_n^+, d_n and d_n^- are the number of input, internal and output variables defined at node n. Therefore, an input variable $U_n^+ \in \mathcal{U}_n^+$ is a matrix of $d_n^+ \times T$ values, $U_n^+ \in \mathbb{R}^{d_n^+ \times T}$. The same holds true for the internal variable $X_n \in \mathcal{X}_n$ and the output variable $U_n^- \in \mathcal{U}_n^-$. In addition, we denote by $h_n^{\leq} : \mathcal{X}_n \times \mathcal{U}_n^+ \times \mathcal{U}_n^- \mapsto \mathbb{R}^{c_n^{\leq}}$ and $h_n^{=} : \mathcal{X}_n \times \mathcal{U}_n^+ \times \mathcal{U}_n^- \mapsto \mathbb{R}^{c_n^{=}}$ the functions used to define inequality and equality constraints at node $n \in \mathcal{N}$, where $c_n^{\leq}$ denotes the number of inequality constraints and $c_n^{=}$ is the number of equality constraints. Finally, we denote by $f_n : \mathcal{X}_n \times \mathcal{U}_n^+ \times \mathcal{U}_n^- \mapsto \mathbb{R}$ the local objective function at node $n \in \mathcal{N}$, and $g_e^{=} : \mathcal{U}_n^+ \times \mathcal{U}_n^- \times \mathcal{U}_m^+ \times \mathcal{U}_m^- \mapsto \mathbb{R}, e = (n, m) \in \mathcal{E}$ a set of functions defining the topology of the system. The full problem is given by the equations:

$$
\begin{aligned}
\min \ & \sum_{n \in \mathcal{N}} f_n(X_n, U_n^+, U_n^-) \\
\text{s.t. } & h_n^{\leq}(X_n, U_n^+, U_n^-) \leq \mathbf{0}, \forall n \in \mathcal{N} \\
& h_n^{=}(X_n, U_n^+, U_n^-) = \mathbf{0}, \forall n \in \mathcal{N} \\
& g_e^{=}(U_m^+, U_m^-, U_n^+, U_n^-) = 0, \forall e = (n, m) \in \mathcal{E} \\
& X_n \in \mathcal{X}_n, U_n^+ \in \mathcal{U}_n^+, U_n^- \in \mathcal{U}_n^-, \forall n \in \mathcal{N}.
\end{aligned}
$$

In this work, we assume the functions f_n, $h_n^{\leq}$, $h_n^{=}$ and $g_e^{=}$ to be affine. Hence, using a matrix representation is convenient.

Therefore, the matrices X_n, U_n^- and U_n^+ can be written as:

$$
\begin{aligned}
X_n &= \begin{pmatrix} \mathbf{x_{n\,1}} & \mathbf{x_{n\,2}} & \dots & \mathbf{x_{n\,d_n}} \end{pmatrix}^\intercal \\
U_n^- &= \begin{pmatrix} \mathbf{u_{n\,1}^-} & \mathbf{u_{n\,2}^-} & \dots & \mathbf{u_{n\,d_n^-}^-} \end{pmatrix}^\intercal \\
U_n^+ &= \begin{pmatrix} \mathbf{u_{n\,1}^+} & \mathbf{u_{n\,2}^+} & \dots & \mathbf{u_{n\,d_n^+}^+} \end{pmatrix}^\intercal
\end{aligned}
$$

where the symbol $^\intercal$ means the matrix is transposed and the elements $\mathbf{x_{n\,i}}$, $\mathbf{u_{n\,i}^-}$ and $\mathbf{u_{n\,i}^+}$ are, respectively, the i^{th} variable defined for all the timesteps. In other words, we can write the vectors $\mathbf{x_{ni}}$, $\mathbf{u_{ni}^-}$ and $\mathbf{u_{ni}^+}$, as follows,

$$
\mathbf{x_{ni}} = \begin{pmatrix} x_{ni}^{t=0} \\ \vdots \\ x_{ni}^{t=T-1} \end{pmatrix}, \qquad \mathbf{u_{ni}^-} = \begin{pmatrix} u_{ni}^{-\,t=0} \\ \vdots \\ u_{ni}^{-\,t=T-1} \end{pmatrix}, \qquad \mathbf{u_{ni}^+} = \begin{pmatrix} u_{ni}^{+\,t=0} \\ \vdots \\ u_{ni}^{+\,t=T-1} \end{pmatrix}
$$

We denote by v_n the total number of variables in node $n \in \mathcal{N}$. We can define one last matrix noted W_n, we call variable matrix, which is the concatenation of the three matrices X_n, U_n^- and U_n^+. The matrix W_n contains all the variables in the node n. It is given by,

$$W_n = \begin{pmatrix} X_n & U_n^+ & U_n^- \end{pmatrix}$$
$$= \begin{pmatrix} x_1^0 & ... & x_i^0 & u_1^{+\,0} & ... & u_j^{+\,0} & u_1^{-\,0} & ... & u_k^{-\,0} \\ x_1^1 & ... & x_i^1 & u_1^{+\,1} & ... & u_j^{+\,1} & u_1^{-\,1} & ... & u_k^{-\,1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_1^{T-1} & ... & x_i^{T-1} & u_1^{+\,T-1} & ... & u_j^{+\,T-1} & u_1^{-\,T-1} & ... & u_k^{-\,T-1} \end{pmatrix}$$

One linear inequality constraint defined by $h_n^{\leq}(X_n, U_n^+, U_n^-)$ can be written as,

$$\sum_{i=0}^{W-1} \sum_{t=0}^{T-1} a_{i\,t} w_{i\,t} - b \leq 0 \tag{6.1}$$

where $a_{i\,t} \in \mathbb{R}\ \forall i, t$ are the coefficient of the linear constraint and $b \in R$ is the independent coefficient. In matrix form, we write this constraint as

$$\text{trace}(A_n W_n) - b \leq 0$$

where $A_n \in \mathbb{R}^{v_n \times T}$ is the matrix of coefficient $a_{i\,t}$ in Equation (6.2) and where the trace is a linear operator corresponding to the following matrix operation:

$$\text{trace}(Y) = \sum \mathbf{s_i} Y \mathbf{s_i^\intercal}$$

where the vector $\mathbf{s_i} = \begin{pmatrix} 0 & \dots & 0 & 1 & 0 & \dots & 0 \end{pmatrix}$ is a vector of zeros with only a single 1 value on the i^{th} component. These vectors are used to retrieve the i^{th} element on the diagonal of the product $A_n W_n$. The equality constraints defined by $h_n^{=}(X_n, U_n^+, U_n^-)$ are expressed as two inequality constraints,

$$h_n^{=}(X_n, U_n^+, U_n^-) <= 0$$
$$h_n^{=}(X_n, U_n^+, U_n^-) >= 0$$

Similarly, the linear objective function $f_n(X_n, U_n^+, U_n^-)$ can be written as,

$$\text{trace}\big(B_n W_n\big) \tag{6.2}$$

with $B_n \in \mathbb{R}^{v_n \times T}$ is the coefficient of each variables for the objective function defined in f_n, build similarly to A_n.

Furthermore, the linking constraint $g_e^{=}(U_m^+, U_m^-, U_n^+, U_n^-)$ associated to each edge $(n, m) \in \mathcal{N}$. can be written as:

$$W_n C = W_m D,\ \forall e = (n, m) \in \mathcal{E}$$

where W_n, W_m respectively are the variable matrices of the node n and m and $C \in \mathbb{R}^{v_n \times q}$ and $D \in \mathbb{R}^{v_m \times q}$ are matrices with each column containing only one element of value one.

6.3 Compiler

In the following section, we explain the compiler's implementation, the choices made and give a further insight into the difficulties faced. A compiler was implemented to translate the problem written in the GBOML language to the formalization explained in the previous section. In the PLY section, we quickly introduce the chosen programming language to build the lexer and parser. Then, we explain each step of the compilation from the input file to the output, going through the lexing, parsing, semantic analysis, intermediate representation and final matrix representation. By dividing the compiler in distinct parts we can easily replace one component without changing everything in the implementation.

6.3.1 PLY

To enable other researchers to takeover this work, we decided to implement the compiler in `Python`. `Python`'s PLY [22] package enables the easy implementation of a compiler's lexer and parser as it implements a replica of the `Flex` and `Yacc` libraries [23] usually used for lexing and parsing. These two libraries offer a great set of features and enable a better reporting of errors. Yacc enables to create a bottom-up parser and Flex enables the implementation of a lexer. The PLY package, even tough more basic in terms of error reporting, was sufficient in our case.

6.3.2 Lexer

The lexer's implementation is quite straightforward once the token defined. Nevertheless, the optimizations made inside the lexer are worth explaining. As explained in Section 4.2.1, the lexer initially builds an NFA that is converted to a DFA. This operation has a time complexity of $O(2^m)$ in the worst-case, where m denotes the number of states in the NFA. Therefore, we minimize the number of states in the NFA by simplifying and combining different token definitions. Indeed, we generalize the accepted expressions of some tokens (accepting more words than the definition allows) and pose on them a temporary token label. Each element with that given label, is later on checked to see if it corresponds to a token of the language, raising an error if not.

For example, our language possess the 20 following keywords :

```
min = "min";              node = "#NODE";              import = "import";
max = "max";              param = "#PARAMETERS";       for = "for";
input = "input";          cons = "#CONSTRAINTS";       and = "and";
output = "output";        var = "#VARIABLES";          mod = "mod";
internal = "internal";    obj = "#OBJECTIVES";         where = "where";
in = "in";                time = "#TIMEHORIZON";       not = "not";
or = "or";                links = "#LINKS";
```

That makes at least 20 states in a NFA just for keywords. It means that, when converted to a DFA, we have 2^{20} states in the worst-case. We can drastically reduce this number of states by finding patterns between the definitions and combining them.

First, we can combine all the keywords starting with a `#` symbol into one automaton. To do so, instead of building an automaton that accepts only our keywords. We build one that accepts any word starting with the symbol `#` and give to him a temporary label : `key`. All the words with that temporary token label are subsequently checked to see if they belong to the list of pre-defined keywords. If they do belong to the list, we replace the temporary token label by the corresponding one (`node`, `param`, `cons`, `var`, `obj`, `time`, `links`). If not, we raise an error. To continue the illustration, the word `#NOT_A_KEYWORD` is accepted by our automaton and given the temporary label `key`. However, since it does not belong to the list of keywords, an error is raised.

Second, we combine the definition of all the other keywords with the one of identifiers. As a reminder the identifier token is defined as:

```
id = letter,{letter,"_",digit,dollar};
```

We give the temporary label of `id` to this combination of the two. If the word read corresponds to one of the keywords, we change the label to the keyword's token, else we keep the `id` token.

By proceeding this way, instead of having 2^{20} states in the DFA for keywords only, we would have 2^1 states in the worst-case. In additon to these modifications, we generalize the filename token by accepting any string between quotation marks and discriminating later. We also combine integers and floats in one definition. The partial equivalent automaton is illustrated in Figure 6.3.

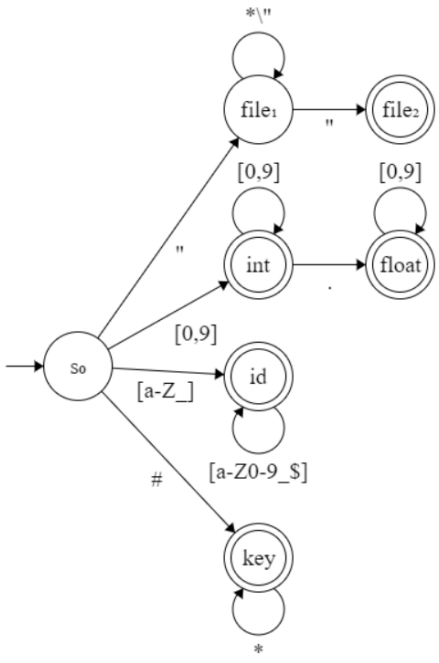

Figure 6.3: Partial deterministic finite automaton built in the lexer for the file, int, float, id and keyword tokens. s_0 is the starting state, file$_2$, int, id, key are all accepting states. The symbol '*' means any character and the symbols '/"' mean except '"'.

This way of proceeding has another advantage, by accepting more and discriminating later, it enables the generation of more precise errors. If we build an automaton that identifies the keyword `"#NODE"` and another that identifies `"#PARAMETERS"`, then, if an input such as `#NOT_A_KEYWORD` is provided, as both automata reject the input, the lexer has no accepting state to go back to and reports an error as:

```
"the character "#" is not defined"
```

which is unclear. Our solution reports the error:

```
"the keyword sequence "#NOT_A_KEYWORD" is not defined"
```

6.3.3 Parser

The parser uses the grammar defined in Section 5.1 and builds an abstract syntax tree represented in Figure 6.4 as an instance of the Program class.

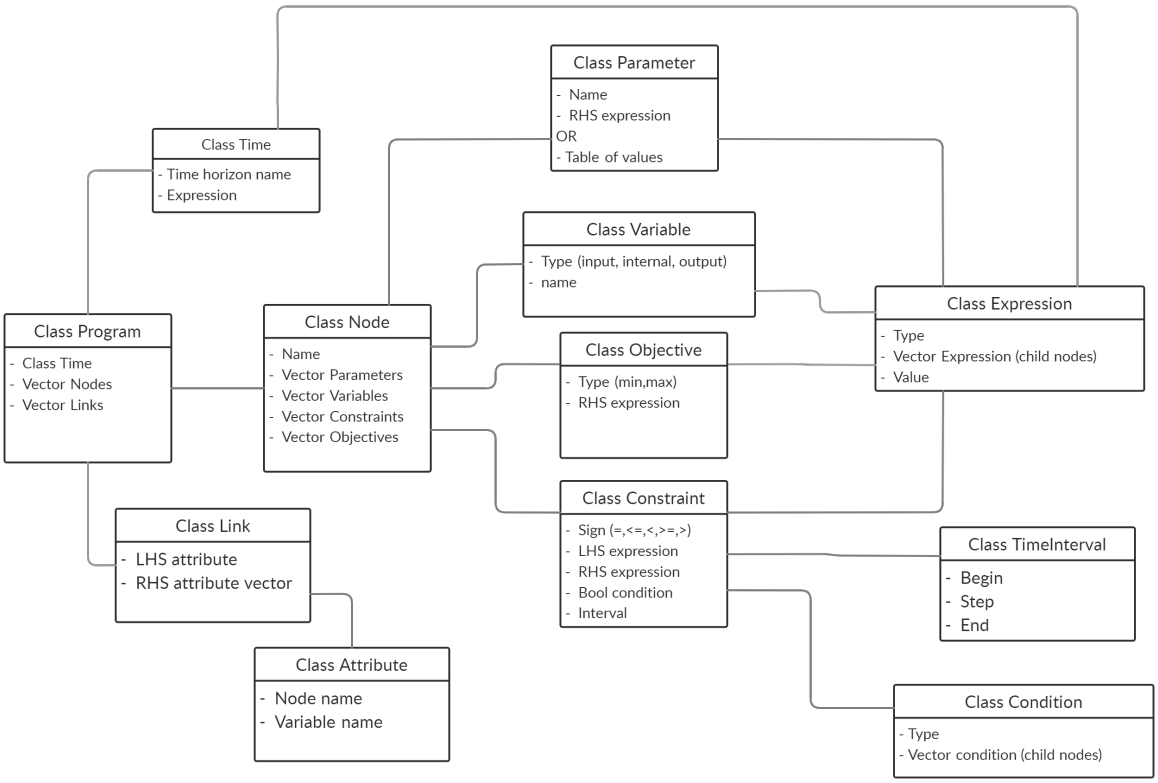

Figure 6.4: Diagram representing the different classes in the abstract syntax tree generated by the parser. A program written in GBOML is an instance of the class Program. Each node corresponds to a class in the tree and the link between two nodes mean that at least one node uses as object an instance of the other class.

The Program class contains an instance of the Time class, a vector of Node objects and a vector of Link objects. It is equivalent to the grammar rule `start = [time], {node}, [link]`. The `time` object keeps the name of the chosen variable for the horizon and the input expression equivalent, `time = id, "=", expression`. The Link class contains a left-hand side instance of the class attribute and a right-hand side vector of instances of attribute. The Attribute class contains a node name and a variable name, `attribute = id, ".", id`.

The Node class is determined by a unique name, a vector of parameters, a vector of variables, a vector of constraints and a vector of objectives, `node = "#NODE" id, [parameter], variable, [variable], [constraint], [objective]`. A parameter is determined by a name and a right-hand side expression, `parameter = id, "=", expression`. A variable is determined by a name and a type among "input", "output" and "internal", `variable = type, ":", identifier`. A constraint is defined by a sign, either `"="`, `"<="` or `">="`, and two expressions, corresponding to the right and left-hand side of the sign, `constraint = expression, sign, expression`. In addition, an interval for which the constraint shall be evaluated and

a condition can be added, `adv_constr = constraint, interval, boolean_condition`. The interval comes in the form of a TimeInterval object containing three integers corresponding to the start, step size and end of the interval. Furthermore, a condition tree can be added, which is determined by its type and condition children nodes.

An objective is determined by a type, either a maximization or a minimization and an expression, `objective = type, ":", expression`. An expression is determined by its type, a vector of child nodes and a value.

The expressions are built as expression tree. The leaves correspond to either an integer, a float or an identifier. This information is contained in the value field of the expression and the leaf's type is by definition "literal". A leaf does not have child nodes. An inner node does not contain a value, its type is determined by the operator of the corresponding operation and its children are the corresponding left and right-hand side of the operation. For example, we consider the expression

$$(3/2 + x) * (5 - 1)$$

the corresponding expression tree is given in Figure 6.5.

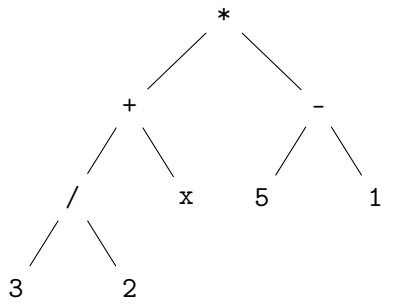

Figure 6.5: Example of expression tree. The expression represented is (3/2+x)*(5-1).

The precedence in the expression maps to a hierarchical relation between operators in the tree. The precedence rules are explained in Subsection 5.3.

The conditions are also built as a tree, similarly to the expressions. However, the internal nodes are boolean operators, namely `AND`, `OR`, `NOT` and also comparison operators : `"<="`, `"<"`, `"=="`, `"!="`, `">="` and `">"`. The leaves of a condition are expressions.

A condition can be,

$$t + 2 > 3 \texttt{ AND } t < 10$$

and is represented in Figure 6.6.

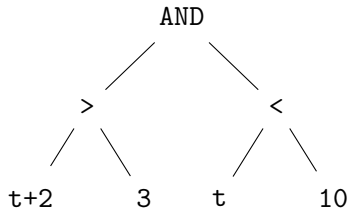

Figure 6.6: Example of boolean condition tree. The illustrated condition is $t+2 > 3$ `AND` $t < 10$.

6.3.4 Semantic Analysis

In the semantic analysis, we check that the previous instance of the Program class respects the semantics of the language. Most errors that can occur are linked to the context as the previous syntax tree is built from a context-free grammar. The main point in this part of the code was to find the relevant errors to report. Some are obvious as :

- checking if variables, parameters have been previously defined,
- checking if there is no redefinition,
- checking that an imported file exists,
- checking that links correspond to the `Node.ouput = list of Node.input`,

to name a few. These are classical errors that we can find in many semantic analysis of common programming languages.

There also are more advanced, less common errors. Typically, as we consider linear expressions with respect to the variables, we needed to go through all the expressions in constraints and objectives to check for linearity. Some of these more advanced errors can only be checked for when we try to generate the internal, node-based mathematical representation. As a result, the semantic analysis and the additional internal representation explained in the Subsection 6.3.5 are intertwined. The errors that can only be checked during or after the internal representation is constructed are:

- checking for every value of `t` in the interval that every index in constraints and objectives is properly defined,
- checking that indexes in parameter definitions are valid,
- checking that at least a valid constraint is defined in the document,
- checking that at least one valid objective exists

All other errors are checked before the construction of the internal representation.

6.3.5 Building the Internal Representation

In this subsection, we discuss the algorithms that convert constraints and objectives into the mathematical formulation described in Section 6.2. In other words, we need to find the coefficient multiplying a particular variable. If we consider again the expression in Figure 6.5, our variable would be `x` and the coefficient we are looking for would be `(5-1)`.

Main Algorithms

Turning expression trees into this mathematical equivalent is the most interesting part of our implementation from an algorithmic perspective. This transformation is the bottleneck of our implementation with respect to the time complexity. Therefore, we give a further analysis. We present the algorithms used for converting constraints in their mathematical equivalent. The exact same algorithms are applied for the conversion of objectives into matrices. Algorithm 1 gives a global overview of the operations performed and is particularly useful for complexity analysis. The Algorithm 1 takes a vector of node objects previously defined and turns their constraints into their matrix form.

We next describe the exact specification of the different functions used in Algorithm 1. The function `retrieve_variables` takes a node as input and returns its list of variables. The function `retrieve_constraints` works similarly but returns the list of constraints. The function `build_variables_matrix` takes a list of variables $[v_1, ..., v_n]$ and builds a matrix where each columns corresponds to a variable and each line to a timestep. More formally, it constructs a matrix of form

$$\begin{pmatrix} v_1[0] & ... & v_n[0] \\ \vdots & & \vdots \\ v_1[T-1] & ... & v_n[T-1] \end{pmatrix}$$

The function `get_leafs` takes a constraint object as input and returns its right-hand side and left-hand side leaves. The functions `retrieve_interval` and `retrieve_condition` take a constraint as input and respectively return its interval of validity and its condition. The two functions `retrieve_index` and `retrieve_name` take as input a leaf containing a variable and respectively output its index expression and its name.

The function `evaluate` takes an expression as input and returns its value. The equivalent algorithm is explained in Algorithm 2. The type of an expression can either be `literal` or the operator applied.

The function `compute_variable_factor_expression` takes an expression and a variable as input and returns the term multiplying the variable. The algorithm of this function is given in Algorithm 4. The function `compute_variable_factor` shown in Algorithm 3, takes a constraint and computes the aforementioned expression equivalent on the right-hand side and left-hand side of the equality and subtracts the two to get the term multiplying a certain variable in a constraint.

Algorithm 1: Turning constraints in their matrix form

```
Input: nodes
output: all_constraints
for node in nodes do
    variables = retrieve_variables(node)
    constraints = retrieve_constraints(node)
    variables_matrix = build_variables_matrix(variables)
    all_constraints = empty list
    for cons in constraints do
        leafs = get_leafs(cons)
        variables_in_leafs = leafs ∩ variables
        interval = retrieve_interval(cons)
        condition = retrieve_condition(cons)
        triplets = empty list
        for t in interval do
            if t satisfies condition then
                for leaf in leafs do
                    current_index = evaluate(retrieve_index(leaf))
                    n,m = find(retrieve_name(leaf),current_index) in variables_matrix
                    value = compute_variable_factor(constraint,leaf)
                    triplets.add(value,n,m)
                end
                independant_term = compute_indep_factor(constraint)
                all_constraints.add(triplet,independant_term,get_sign(cons))
            end
        end
    end
end
```

If we note by N, C, T and L respectively the number of nodes, the maximum number of constraint in one of the nodes, the time horizon and the maximum number of leafs in one of the constraints. In the worst-case scenario, Algorithm 1 executes $2 * N * C * T * L$ times the Algorithm 4, one for each expression of each side of the constraint sign. Algorithm 4 has the same time complexity as a tree exploration $O(M)$ where M is the number of nodes in the expression tree. First, Algorithm 4 needs to find the variable inside the expression tree which corresponds to a regular tree exploration. Second, when the variable is found, it evaluates the other branch of the parent node in case of multiplication or division which correspond to an additional partial exploration of the tree. As we have a full exploration $O(M)$ in complexity and a partial exploration whose complexity is lower than $O(M)$. The full time complexity is at most $O(M) + O(M)$ which is still a linear complexity $O(M)$. An example of worst-case scenario is illustrated in Figure 6.7, where we have to re-explore and evaluate all the second branches. In this case, we explore the tree of $M = 15$ nodes. The number of exploration E is

Algorithm 2: evaluate

```
Input: expression
output: value
children = retrieve_child(expression)
type = retrieve_type(expression)
if type is literal then
    value = retrieve_value(expression)
else
    values = empty vector
    i =0 for child in children do
        values[i] = evaluate(child)
        i = i+1
    end
    value = apply type to values
end
```

Algorithm 3: compute_variable_factor

```
Input: constraint
Input: variable
output: value
rhs = retrieve_rhs(constraint)
value1 = compute_variable_factor_expression(rhs,variable)
lhs = retrieve_lhs(constraint)
value2 = compute_variable_factor_expression(lhs,variable)
value = value1 - value2
```

Algorithm 4: compute_variable_factor_expression

```
Input: expression
Input: variable
children = retrieve_child(expression)
value = 0
found = False
if expression is literal and expression is variable then
    value = 1
    found = True
else if expression is literal and expression is not variable then
    value = 0
    found = False
else
    type_op = retrieve_type(expression)
    value1,found = compute_variable_factor_expression(child[0],variable)
    if two children then
        if found then
            if type_op is multiplication or division then
                value2 = evaluate(child[1])
            else if type_op is addition or subtraction then
                value2,found = compute_variable_factor_expression(child[1],variable)
            value = value1 type_op value2
        else
            value2,found = compute_variable_factor_expression(child[1],variable)
            if found and type_op is multiplication then
                value1 = evaluate(child[0])
                value = value1 type_op value2
        end
    else if one child then
        value = type_op value1
end
```

given by :

$$E = M + \frac{M+1}{2} - 1 + \frac{M+1}{4} - 1 + \frac{M+1}{8} - 1 = 26 < 2M$$

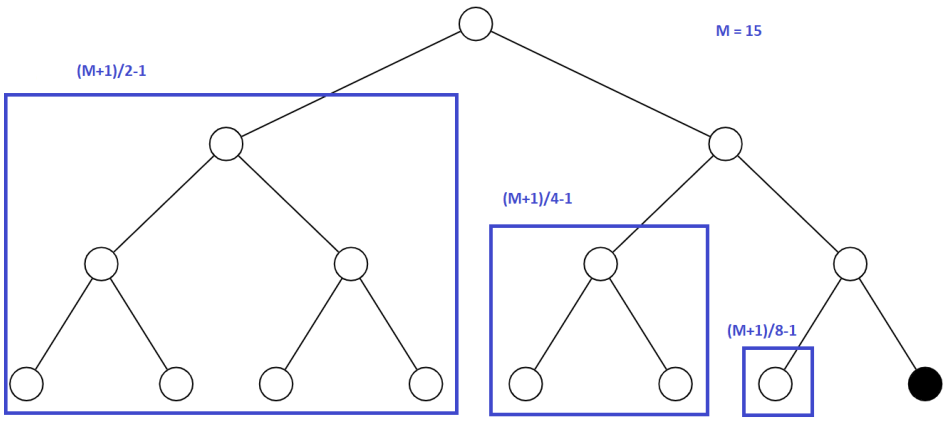

Figure 6.7: worst-case scenario of Algorithm 4 where at each step there is reevaluation of the other branch. The number of notes is denoted by N. The black node is the variable found in the tree.

Therefore, the full time complexity of our implementation to turn constraints in their matrix form is $O(N * C * T * L * M)$. In addition, if we note by B the maximum number of objectives in one of the nodes, the time complexity of turning objectives into their matrix form is $O(N * B * T * L * M)$.

The Algorithm 1 could be strictly limited to one full exploration if we combined the evaluation and the exploration part in one big algorithm. However, as the evaluation might be changed in the future with further functionalities we decided to centralize it in one singular function and to reuse it everywhere. We therefore pay the cost of going through some nodes twice. It also enables an easier maintenance of the code.

Advanced Implementation

Algorithm 1 gives the basic algorithmic structure. However, in practice, we take into account several other features so we do not recompute the same things several times. For example, we consider the variable x and a horizon of 8760, the constraint,

$$x[0] + 2x[1] = 2$$

is only concerned with the fixed timesteps. Computing this constraint for all values of t in the horizon, $t \in [0, T-1]$, does not make sense. Furthermore, writing T times the same constraint is not efficient in terms of space as they all contain the exact same information. In addition, in terms of computation, we compute the exact same values T times. To deal with this issue, a predicate to check whether a constraint depends on t was implemented. It explores the tree

of expressions and indexes to look for a variable t if there is no t. Then, it only computes the values once and adds them to the constraint matrix.

Another interesting part of the algorithm is how to deal with indexes that reach outside the horizon. Typically, we can consider the variable x with a horizon of 2, the constraint

$$x[t * 20] + x[t] = 5$$

is evaluated for each timestep as,

$$\begin{aligned} x[0] + x[0] &= 5 \\ x[20] + x[1] &= 5 \end{aligned}$$

The second constraint is outside the horizon scope. Therefore, the find operation inside the variables matrix returns False and the constraint is discarded for that particular timestep and the next one is considered. We can not skip all the next timesteps, as we could have for a horizon of 3,

$$x[t-1] + x[t+1] = 0$$

which results in the constraints,

$$\begin{aligned} x[-1] + x[1] &= 0 \\ x[0] + x[2] &= 0 \\ x[1] + x[3] &= 0 \end{aligned}$$

with only the second constraint being a valid one. These results in the fact that,

$$x[t-1] + x[t] \not\equiv x[t] + x[t+1]$$

due to these border effects, as mentioned before.

The best case scenario could be diminished further, by only computing the coefficient of each variable only once if the coefficient is independent of `t`. This would reduce the complexity of best case scenario by a factor `T`.

6.3.6 General Matrix Generation

To be able to retrieve a solution from the internal representation, we decided to generalize the representation. Hence, the last step of the compilation is the general matrix generation. In this step, the compiler takes the internal representation, flattens it (remove the graph strucucture) and converts it to a more general formulation. The output format was chosen as general as it gets to enable an easier connection with any solver. The problem is converted to

$$\begin{aligned} Ax &\leq b \\ \min &: C^T x \end{aligned}$$

We output the matrices A, b and C as they define the problem and are necessary for the solver. However, the solver does not have any information on the variable names or the structure of the input as it only receives the matrices A, b and C^T. It only returns a vector x of values that solve the optimization problem if feasible. Therefore, the compiler also outputs the variable names corresponding to each index to retrieve the information necessary for the communication interface.

6.4 Solver Communication

The communication is composed of two parts. Each part is assigned with a specific task. The chosen solver is `Gurobi`.

The first part is the solver communication. To communicate with the solver, we implemented a communication through `Julia`'s `JuMP`. It then sends the matrices to `Gurobi`. `Gurobi` solves the problem and the solution is sent back to the post-processing part.

The second part is the post-processing. It receives the solution as a raw vector and converts it back to a structure similar to the one entered in the input. It associates each index of the vector with a variable and each variable with a node to enable a better understanding of the solution.

Chapter 7

Discussion and Results

Chapter abstract: *In this chapter we discuss how to encode a small problem in GBOML and discuss its results when given to a solver. We end the discussion by having a look at the evolution of the time taken to generate the matrices for that problem with respect to the horizon.*

In this chapter, the GBOML is leveraged to tackle the optimal design of a solar off-grid microgrid. We benchmark our solution to a `Julia` implementation of the same problem. Then, we perform an empirical analysis of the evolution of the computation time to generate the matrix form of the problem with respect to the horizon.

7.1 Microgrid

The optimal design of a solar off-grid microgrid, presented in Bolland et al [24], consists in finding the optimal capacity of PV panels and the optimal battery capacity to satisfy a certain demand while minimizing the total cost of the microgrid (i.e., the sum of investment and operating costs). Note that this equivalent to minimizing the electricity cost for the microgrid owner. To illustrate, let's say we have a very small factory. We know approximately how much electricity we need per hour each day and we want to minimize our total bill for the next three years. We need to decide whether investing in PV panels and a battery is interesting for us and if it is in how much capacity do we need. The system is made-up of three distinct entities: PV panels, a battery and the factory consumption. A natural way of modeling this problem is to consider each entity as a node and add a meeting point node called the operation. We consider a horizon of 2 years, equivalent to 17520 hours.

Time

The time horizon of 2 years can therefore be written in the language as:

```
1 #TIMEHORIZON T=2*24*365
```

PV Panels

PV panels convert sunlight into electricity. PV panels are limited by their installed capacity, denoted by $\bar{P}^{PV} \in \mathbb{R}^+$ (in watt), which correspond to the maximum quantity of electricity that they can produce. This installed capacity is associated to a marginal cost π^{PV} (in currency/watt). That investment is taken with an interest rate r over a 2 year period, $n = 2$. Therefore, the investment cost is given by:

$$I^{PV} = \bar{P}^{PV} \cdot \pi^{PV} \cdot \frac{(1+r)^n}{(1+r)^n - 1} .$$

The amount of electricity produced is directly proportional to the installed capacity and a factor corresponding to the amount of radiation emitted by the sun. We note that factor p_t^{PV}. Hence, the amount of electricity produced is also influenced by the hour $h \in \{0, ..., 23\}$ of the day. Therefore, the electricity produced, P_t^{PV}, is given by the equality,

$$P_t^{PV} = p_t^{PV} \cdot \bar{P}^{PV} .$$

In this case, we consider a simplify cost model and neglect operating costs for PV panels. The objective of this node is to minimize its investment cost. This can be expressed in the GBOML as:

```
#NODE SOLAR_PV
#PARAMETERS
n = 2
rate = 0.07
pi = 1
irradiance_per_hour = import "mean_irradiance.csv"
#VARIABLES
internal: capacity
internal: investment
output: electricity
#CONSTRAINTS
capacity[0] >= 0;
capacity[0] = capacity[t];
electricity[t] = irradiance_per_hour[mod(t, 24)] * capacity[t];
investment[t] = capacity * rate *
                ((1+rate)**n) / (((1+rate)**n) - 1);
#OBJECTIVES
min: investment[t]/T
```

Battery

A battery is an energy storage device. Similarly to the panel capacity, the energy that the battery can store is limited by its storage capacity, $\bar{E}^B \in \mathbb{R}^+$ (in watt-hour). The overnight investment cost (in currency/watt-hour) is denoted by π^B. We consider a similar rate over the same period. Therefore, the actual investment cost is given by

$$I^B = \bar{E}^B \cdot \pi^B \cdot \frac{(1+r)^n}{(1+r)^n - 1} .$$

The battery can either store energy and release it at a later stage. We denote the charging power by $P_t^{B+} \in \mathbb{R}^+$ (in watt) and the discharging power by $P_t^{B-} \in \mathbb{R}^+$ (in watt). The quantity of energy stored in the battery defines the state of charge. The state of charge is denoted by $SoC_t^B \in \left[0, \bar{E}^B\right]$ (in watt-hour). The constraint linked with the state of charge is:

$$SoC_{t+1}^B = SoC_t^B + \eta \cdot P_t^{B+} - \frac{P_t^{B-}}{\eta} .$$

the cost of deploying and operating the system. Note that in this case, operating costs are assumed to be equal to 0. The objective of this node is also to minimize its investment cost. This node can be written as:

```
#NODE INVEST_BAT
#PARAMETERS
pi = 1
rate = 0.07
n = 2
eta = 1
#VARIABLES
output: investment
internal: capacity
internal: soC
input: charge
output: discharge
#CONSTRAINTS
capacity >= 0;
capacity[0] = capacity;
charge[t] >= 0;
discharge[t] <= 0;
investment[t] = pi * capacity[t] * rate *
                ((1+rate)**n) / (((1+rate)**n) - 1);
soC[0] = capacity / 2;
soC[t] >= 0;
soC[t] <= capacity[t];
soC[t+1] = soC[t] + eta * charge[t]- discharge[t]/eta;
#OBJECTIVES
min: investment[t]/T
```

Factory Consumption

The electricity needed by the factory, denoted by $P_t^C \in \mathbb{R}^+$ (in watt). In GMOBL, we can write:

```
#NODE FACTORY
#PARAMETERS
electricity = import "electricity_per_hour.csv"
#VARIABLES
output: consumption
```

```
#CONSTRAINTS
consumption[t] = electricity[mod(t, 24)];
```

Power Balance

The last node is the operation where we can either consume the electricity from our microgrid. The electricity from the PV panels, P_t^{PV}, the power charged in the battery, P_t^{B-}, are the inputs of the operation node as they provide electricity. The consumption P_t^C is also provided as input. The power we take to charge the battery is the output, P_t^{B-}. We note the power shortage in the microgrid at a time step t as $P_t^{shed} \in \mathbb{R}^+$ (in watt). Formally it is given by,

$$P_t^{shed} = P_t^{B+} + P_t^C - P_t^{PV} - P_t^{B-} \ .$$

This shortage of power is therefore penalized at a marginal price π^{shed} (in currency/watt). The operating cost O_t is given by:

$$O_t = P_t^{shed} \cdot \pi^{shed} \tag{7.1}$$

The operation's objective is to minimize its cost. Therefore, the node is given by:

```
#NODE POWER_BALANCE
#PARAMETERS
pi = 1
#VARIABLES
input: pv_production
input: battery_discharge
output: battery_charge
output: consumption
internal: shed
#CONSTRAINTS
shed[t] >= 0;
shed[t] = battery_charge[t] + consumption[t] - pv_production[t] -
    battery_discharge[t];
#OBJECTIVES
min: shed[t] * pi
```

Linking

The final step needed is to link these nodes between them:

```
#LINKS
SOLAR_PV.electricity = POWER_BALANCE.pv_production
BATTERY.discharge = OPERATION.battery_discharge
FACTORY.consumption = OPERATION.consumption
OPERATION.battery_charge = BATTERY.charge
```

Final Objective

The global objective of our implementation, therefore, is :

$$\min : \sum_{t=0}^{T-1} O_t + I_t^B + I_t^{PV}$$

Results

The results are directly influenced by the electricity consumption and the average per day solar radiations we consider. To obtain some results, we considered the toy values in Appendix .1.2. Without a battery and PV panels, we would pay 56356 currency each year of penalty from the missing production. That makes 112712 currency for two years. By adding the PV panels and the battery, we would pay $34478,82378$ currency for two years. To obtain this, we would need to install a PV capacity of 169.6631 watts per hour and a battery capacity of 114.9868 watts per hour. The power balance for the four first days is shown in Figure 7.1.

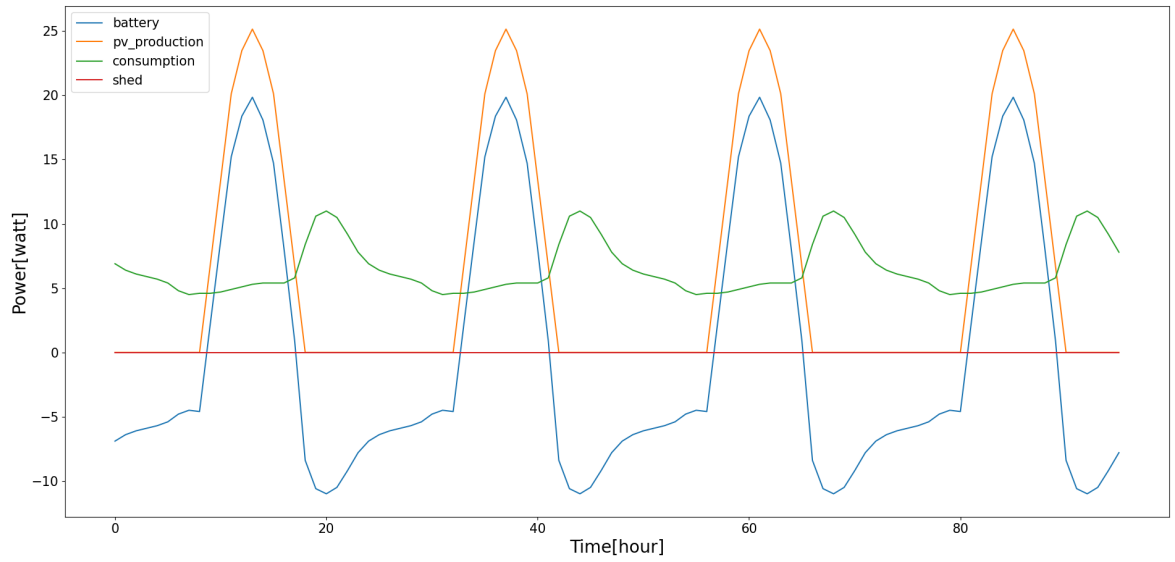

Figure 7.1: Power balance of the microgrid for the first four days. Battery is the balance between consumption and production of the battery.

7.1.1 Benchmarking

To benchmark our proposed solution, we implemented the equivalent problem in `Julia` using `JuMP` and compared the two solutions. These solutions are very different in their respective structure as there is no graph implementation in `JuMP`. However, the exact same results were obtained as illustrated in Figure 7.2. As we go through `JuMP` to generate our solution we can not outperform the `JuMP` equivalent solution in terms of time. The `JuMP` implementation can be found in the delivered files.

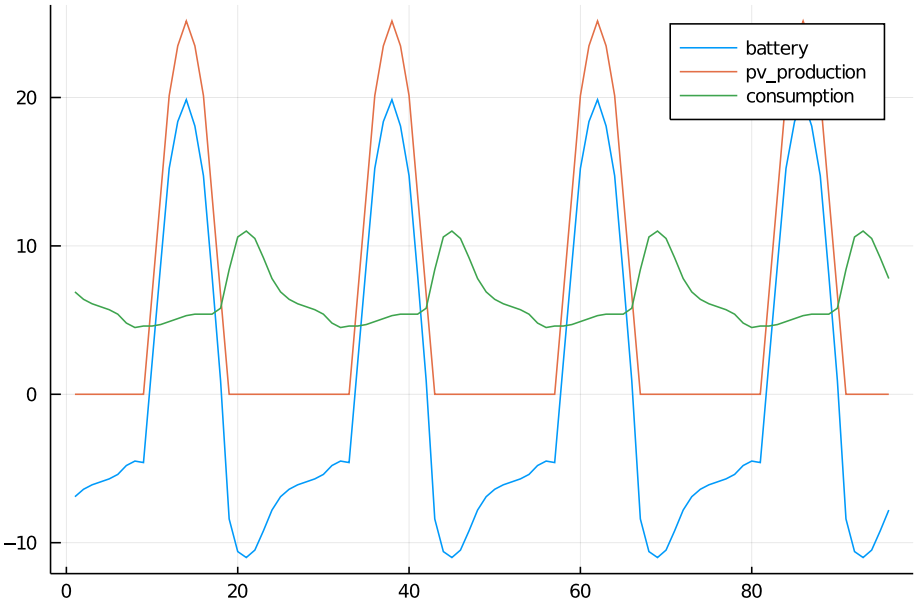

Figure 7.2: Power balance of the microgrid done in `JuMP` for the first four days. Battery is the balance between consumption and production of the battery.

7.1.2 Empirical Complexity Analysis

Algorithm 1 is the bottleneck of our implementation. Therefore, in Section 6.3.5 we concluded that the overall complexity of our solution is $O(N * (C + B) * T * L * M)$ where N is the number of nodes, C the maximum number of constraints defined in one node, B the maximum number of objectives in one node, T the number of values in the interval, L the number of variables in the leaves of a constraint and M the number of nodes in the constraint tree. In the following, we study the evolution of the time complexity of the Microgrid example with respect to the horizon (i.e., we seek to check whether the theoretical complexity bound obtained earlier is tight).

We evaluated the matrix generation for the microgrid example for horizons going from 0 to 1000 with steps of 100. Then, we studied the horizon from 10 000 to 100 000 and used steps of 10000. As expected, we are linear with respect to the time horizon as can be seen in Figure 7.3.

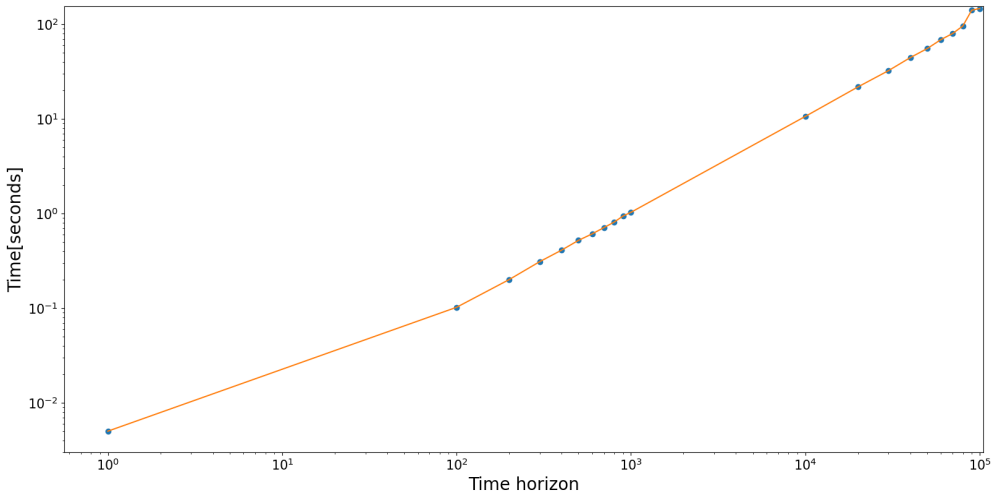

Figure 7.3: Evaluation of the time to generate the matrix representation with respect to the time horizon for the microgrid example.

Chapter 8

Conclusion

The encoding of an optimization problem using a modeling language is an essential part in the workflow of optimization practitioners. The dominant paradigm for optimization modeling languages, algebraic modeling languages, offers an easy implementation of the different classes of optimization problems with an encoding close of the mathematical formulation. However, these languages are ill-suited for exploiting the structure that may exist in certain classes of optimization problems, particularly so for problems involving discrete-time dynamical systems and exhibiting a block structure that can be encoded by a sparse connected graph.

Therefore, we implemented a new language, the Graph-Based Modeling Optimization Language (GBOML), which provides basic functionalities comparable to those found in AMLs, but also natively supports the encoding of problems involving discrete-time dynamical systems and possessing a natural block structure. GBOML facilitates the encoding of such problems for a wide range of practitioners, and can also simplify their postprocessing. To be able to retrieve a solution for a problem in GBOML, we implemented a multi-step solution made of a compiler and a two way link, towards a solver, sending the matrices to the solver to obtain a solution and backwards, to retrieve the solution, convert it in the initial graph structure and assign its optimal value to each variable.

We tested our solution on an energy systems planning problem. More precisely, we benchmarked it with the equivalent problem in expressed in `JuMP` and obtained the exact same results. In addition, we provided an empirical analysis of the time complexity of model generation in GBOML.

The following directions would be worth pursuing in future work. Concerning the features of the language, enabling the declaration of scalar variables and the declaration of empty nodes that work as functions to which the user can provide different parameters without redefining the full node can be a great addition to the language. Extending the class of problem GBOML deals with to non-linear optimizations or mixed integer linear programming can also be an interesting direction to explore by changing the internal representation. One last exploration idea we would like to mention, is to investigation the possibility of using reinforcement learning

to solve problems expressed in GBOML.

Bibliography

[1] M. Berger, D. Radu, R. Fonteneau, T. Deschuyteneer, G. Detienne, and D. Ernst, "The role of power-to-gas and carbon capture technologies in cross-sector decarbonisation strategies," *Electric Power Systems Research*, vol. 180, p. 106039, 2020.

[2] S. H. Low and D. E. Lapsley, "Optimization flow control. i. basic algorithm and convergence," *IEEE/ACM Transactions on Networking*, vol. 7, no. 6, pp. 861–874, 1999.

[3] S. E. Dan Gusfield and C. Langley, "Optimal, efficient reconstruction of phylogenetic networks with constrained recombination," *IEEE/ACM Transactions on Networking*, vol. 2, pp. 173–213, 2004.

[4] D. Gusfield, K. Balasubramanian, and D. Naor, "Parametric optimization of sequence alignment," *Algorithmica*, vol. 12, pp. 312–326, 1992.

[5] J. Choi, H. Lee, S. Heo, and J. Lee, "A mathematical programming for supply chain network design," in *2006 SICE-ICASE International Joint Conference*, pp. 170–174, 2006.

[6] F. Sayyady and S. D. Eksioglu, "Optimizing the use of public transit system during no-notice evacuation of urban areas." http://www.sciencedirect.com/science/article/pii/S0360835210001464, 2010. Accessed: 2021-01-01.

[7] R. Beal, G. Chalkiadakis, T. J. Norman, and S. D. Ramchurn, "Optimising game tactics for football," 2020.

[8] A. Singh, "Optimal solution strategy for games," *International Journal on Computer Science and Engineering*, vol. 2, 12 2010.

[9] J. April, F. Glover, J. Kelly, and M. Laguna, "Simulation/optimization using real-world applications," *Winter Simulation Conference Proceedings*, vol. 1, pp. 134 – 138 vol.1, 02 2001.

[10] M. Berger, D. Radu, A. Richel, and D. Ernst, "Remote renewable hubs for synthetic fuel production," 2020.

[11] J. Kallrath, "Modeling languages in mathematical optimization ," 2004, first edition.

[12] W. E. Hart, J.-P. Watson, and D. L. Woodruff, "Pyomo: modeling and solving mathematical programs in python," *Mathematical Programming Computation*, vol. 3, no. 3, pp. 219–260, 2011.

[13] I. Dunning, J. Huchette, and M. Lubin, "Jump: A modeling language for mathematical optimization," *SIAM Review*, vol. 59, no. 2, pp. 295–320, 2017.

[14] R. Fourer, D. M. Gay, and B. W. Kernighan, "A modeling language for mathematical programming," 2003, second edition.

[15] "General algebraic modeling system." https://www.gams.com/. Accessed: 2020-12-01.

[16] "Advanced Interactive Multidimensional Modeling System." https://www.aimms.com/. Accessed: 2020-12-01.

[17] J. Rivera, "Modeling with extend," in *Proceedings of the 30th Conference on Winter Simulation*, WSC '98, (Washington, DC, USA), p. 257–262, IEEE Computer Society Press, 1998.

[18] P. Wolper, "Introduction à la calculabilité," 2006, third edition.

[19] "Powerset construction for converting a non-determinitic finite automaton into deterministic finite automaton." https://en.wikipedia.org/wiki/Powerset_construction. Accessed: 2020-12-20.

[20] P. Fontaine and C. Soldani, "INFO0085 : Compilers course," 2020.

[21] R. S. Alfred V. Aho, Monica S. Lam and J. D. Ullman, "Compilers: Principles, Techniques, & Tools," 2006, second edition.

[22] "Ply python lex and yacc." https://www.dabeaz.com/ply/. Accessed: 2020-12-20.

[23] T. Mason and D. Brown, *Lex & Yacc*. USA: O'Reilly & Associates, Inc., 1990.

[24] A. Bolland, I. Boukas, F. Cornet, M. Berger, and D. Ernst, "Learning optimal environments using projected stochastic gradient ascent," 2020.

.1 Appendix

.1.1 List of Operators In GBOML

Operator	Meaning
`**`	exponential
`-`	Minus
`+`	Plus
`/`	divided
`*`	times
`=`	equal
`<=`	lower or equal
`>=`	bigger or equal
`<`	strictly lower
`>`	strictly bigger
`!=`	different comparison
`==`	equality comparison
`NOT`	NOT predicate
`AND`	AND predicate
`OR`	OR predicate

Table 1: list of GBOML operators.

.1.2 Microgrid Values

hour	Factory Consumption (watt)	Radiations Factor (percentage)
0	6.9	0
1	6.4	0
2	6.1	0
3	5.9	0
4	5.7	0
5	5.4	0
6	4.8	0
7	4.5	0
8	4.6	0
9	4.6	0.04
10	4.7	0.08
11	4.9	0.12
12	5.1	0.14
13	5.3	0.15
14	5.4	0.14
15	5.4	0.12
16	5.4	0.08
17	5.8	0.04
18	8.4	0
19	10.6	0
20	11.0	0
21	10.5	0
22	9.2	0
23	7.8	0

Table 2: Values considered for the simulation of the Off grid problem.