Graph-Based Optimization Modeling Language: A Tutorial

Mathias Berger Adrien Bolland Bardhyl Miftari Hatim Djelassi
Damien Ernst

February 10, 2021

This paper introduces the graph-based optimization modeling language (GBOML), which enables the easy implementation of a broad class of structured linear programs typically found in applications ranging from energy system planning to supply chain management. More precisely, the language is particularly well-suited for representing problems involving the optimization of discrete-time dynamical systems over a finite time horizon and possessing a block decomposable structure that can be encoded by a sparse connected graph. The language combines elements from both algebraic and object-oriented modeling languages in order to facilitate problem encoding and post-processing. This document discusses the abstract problem class that can be represented using the modeling language, details its grammar and provides two relevant examples of applications. The first example deals with the deployment of a microgrid system, while the second example focuses on the design and analysis of remote carbon-neutral fuel supply chains.

1 Introduction

Algebraic modeling languages (AMLs) are the dominant class of modeling languages currently used to encode and implement mathematical programming models [5, 9]. In AMLs, algebraic expressions involving parameters and variables form the basic blocks based on which the objective function and the constraints are defined. Most languages also provide functionalities enabling the definition of vectorized or indexed expressions, which simplifies the construction of large-scale models. Hence, optimization problems must typically be written in an index-based formulation that closely resembles standard mathematical notation, resulting in compact, monolithic models. From a practical standpoint, making proper use of the block decomposable structure a complex model may possess can bring about notable benefits (e.g., the encoding of a model in a modular and decentralised manner, possibly by different modellers, and the assembling of different blocks at a later stage). Although the indexing capabilities of AMLs are often used to encode some structure in the problem (e.g., the topology of the underlying network in power system applications may be represented via node and edge indices), they are usually ill-suited for exploiting the block structure a model may exhibit.

By contrast, the so-called object-oriented modeling paradigm has long been in use in the field of complex systems simulation [9]. Prime examples of this approach include object-oriented modeling languages (OOMLs) such as Modelica and MATLAB Simulink. In this context, models are typically constructed block by block, and all blocks are assembled afterwards in order to build a complete and consistent model. This approach is particularly useful in cases where each block is connected to a relatively small number of other blocks. In the jargon of graph theory, assuming that a block is viewed as a node, this implies that the structure of such models may be represented by a sparse connected graph. This property can be directly exploited to build models in a modular way and simplify their encoding. The ability to build models in such fashion may also have advantages for postprocessing and may facilitate the use of decomposition algorithms [3] and specialised solvers [6] exploiting the underlying problem structure.

This paper introduces the graph-based optimization modeling language (GBOML), which is geared towards the representation of linear programs 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. GBOML draws on concepts from both OOMLs and AMLs to facilitate problem encoding and post-processing. More specifically, it relies on a graph abstraction of optimization problems wherein nodes represent blocks (that correspond to subproblems) with their own parameters, variables, constraints and local objective, while edges represent the relationships between nodes via equality contraints involving variables of nodes to which they are incident. In addition, it makes use of a global time horizon and associated set of time periods that are common to all nodes. Variables are defined for each time period, and constraints can be defined for subsets of time periods constructed using algebraic expressions and logical conditions. A parser for GBOML, called the GBOML compiler, has also been implemented [7, 8], and directly interfaces with linear programming solvers, namely Gurobi, CPLEX and Clp.

This document is structured as follows. Section 2 presents the abstract problem class that can be represented in the modeling language. Section 3 introduces the modeling language along with the associated grammar, and details its basic components. Then, Section 4 illustrates the modeling language using a set of examples, including a microgrid sizing and operation problem and a remote carbon-neutral fuel supply chain design problem. Finally, Section 5 concludes the document.

2 The Abstract GBOML Problem

The modeling language is particularly well-suited for representing 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 [1]. To this end, a global discretised time horizon (and an associated set of time periods) common to all nodes is defined. Nodes model blocks, which can be viewed as optimization subproblems, while edges express the relationships between nodes and their associated variables. More precisely, each node is equipped with a set of internal, input, output variables. Each one of these variables is a vector variable whose entries correspond to different time periods. A set of constraints is also defined for each node, along with a local objective function representing its contribution to a system-wide objective. Finally, for each edge, equality constraints involving the input and output variables of the nodes to which the edge is incident are defined in order to express the relationships between nodes. In the following paragraphs, we formally define variables, constraints, objectives and formulate the abstract model that encapsulates the class of problems considered.

Let T be the time horizon considered, let $\mathcal{T} = \{0, 1, \dots, T-1\}$ be the associated set of time periods, and let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be the undirected graph encoding the block structure of the problem considered, with node set $\mathcal{N}$ and edge set $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$. Let I_x^n, I_u^n and I_y^n be the number of internal, input and output variables defined per time period at node $n \in \mathcal{N}$, and let $\mathcal{I}_x^n = \{1, \dots, I_x^n\}$, $\mathcal{I}_u^n = \{1, \dots, I_u^n\}$ and $\mathcal{I}_y^n = \{1, \dots, I_y^n\}$ be the associated index sets. Now, let $x_i^n \in \mathbb{R}^T$, $\forall i \in \mathcal{I}_x^n$, $u_i^n \in \mathbb{R}^T$, $\forall i \in \mathcal{I}_u^n$, and $y_i^n \in \mathbb{R}^T$, $\forall i \in \mathcal{I}_y^n$, be the set of internal, input and output variables defined at node $n \in \mathcal{N}$, and let $X^n \in \mathcal{X}^n \subseteq \mathbb{R}^{I_x^n \times T}$, $U^n \in \mathcal{U}^n \subseteq \mathbb{R}^{I_u^n \times T}$ and $Y^n \in \mathcal{Y}^n \subseteq \mathbb{R}^{I_y^n \times T}$ be matrix variables obtained by concatenating these internal, input and output variables. Let u_{it}^n and y_{jt}^n denote the i^{th} input variable and the j^{th} output variable at time $t \in \mathcal{T}$ and node $n \in \mathcal{N}$, respectively.

Both equality and inequality constraints may be defined at each node $n \in \mathcal{N}$. More precisely, an arbitrary number of constraints that can each be expanded over a subset of time periods may be defined. Hence, we consider equality constraints of the form

$$h_k^n(X^n, U^n, Y^n, t) = 0, \ \forall t \in \mathcal{T}_k^n, \tag{1}$$

with (scalar) affine functions h_k^n and index sets $\mathcal{T}_k^n \subseteq \mathcal{T}$, $k = 1, \dots, K^n$, as well as inequality constraints

$$g_k^n(X^n, U^n, Y^n, t) \leq 0, \ \forall t \in \bar{\mathcal{T}}_k^n, \tag{2}$$

with (scalar) affine functions g_k^n and index sets $\bar{\mathcal{T}}_k^n \subseteq \mathcal{T}$, $k = 1, \dots, \bar{K}^n$.

Now, let $F^n : \mathcal{X}^n \times \mathcal{U}^n \times \mathcal{Y}^n \to \mathbb{R}$ denote the function defining the objective at node $n \in \mathcal{N}$. In this

document, we consider local objectives of the form

$$F^n = \sum_{t \in \mathcal{T}} f^n(X^n, U^n, Y^n, t),$$

where, for each $t \in \mathcal{T}$, f^n is an affine function of X^n, U^n and Y^n.

For each edge $e = (n, n') \in \mathcal{E}$, a set of scalar equality constraints links either an input variable of node $n \in \mathcal{N}$ to an output variable of node $n' \in \mathcal{N}$ or an output variable of node $n \in \mathcal{N}$ to an input variable of node $n' \in \mathcal{N}$ for all time periods. Thus, for variables $i \in \mathcal{I}_u^n$ and $j \in \mathcal{I}_y^n$ (or vice-versa), these constraints can be expressed as

$$u_{it}^n = y_{jt}^{n'}, \ \forall t \in \mathcal{T}, \text{ or } y_{it}^n = u_{jt}^{n'}, \ \forall t \in \mathcal{T}.$$

Furthermore, let $H^e : \mathcal{U}^n \times \mathcal{Y}^n \times \mathcal{U}^{n'} \times \mathcal{Y}^{n'} \to \mathbb{R}^T$ be an affine function such that the equality constraint associated with each edge $e = (n, n') \in \mathcal{E}$ can be compactly expressed as

$$H^e(U^n, Y^n, U^{n'}, Y^{n'}) = 0.$$

Finally, using this notation, the class of problems that can be represented in this framework reads

$$\begin{aligned} \min \quad & \textstyle\sum_{n \in \mathcal{N}} \left[\sum_{t \in \mathcal{T}} f^n(X^n, U^n, Y^n, t) \right] \\ \text{s.t.} \quad & h_k^n(X^n, U^n, Y^n, t) = 0, \ \forall t \in \mathcal{T}_k^n, \ k = 1, \dots K^n, \ \forall n \in \mathcal{N} \\ & g_k^n(X^n, U^n, Y^n, t) \leq 0, \ \forall t \in \bar{\mathcal{T}}_k^n, \ k = 1, \dots \bar{K}^n, \ \forall n \in \mathcal{N} \\ & H^e(U^n, Y^n, U^{n'}, Y^{n'}) = 0, \ \forall e = (n, n') \in \mathcal{E} \\ & X^n \in \mathcal{X}^n, U^n \in \mathcal{U}^n, Y^n \in \mathcal{Y}^n, \ \forall n \in \mathcal{N}. \end{aligned} \tag{3}$$

In this document, it is also assumed that $\mathcal{X}^n = \mathbb{R}^{m_x^n \times T}$, $\mathcal{U}^n = \mathbb{R}^{m_u^n \times T}$ and $\mathcal{Y}^n = \mathbb{R}^{m_y^n \times T}$. In other words, the problems that can be represented in the modeling language form a class of structured linear programs.

3 The GBOML Grammar

In order to encode and implement an instance of the abstract model discussed in Section 2, an input file written in the GBOML grammar can be parsed by the GBOML compiler. The structure of the input file corresponds directly to the structure of the abstract problem model. Indeed, the input file is structured into blocks, where the time horizon information is given in the first block, each node corresponds to one further block, and the connectivity of the graph abstraction of the model is given in the final block. Each block is introduced by one of the keywords `#TIMEHORIZON`, `#NODE`, and `#LINKS`. Then, an input file is structured as follows.

```
#TIMEHORIZON
// time horizon definition

#NODE <identifier>
// first node definition

#NODE <identifier>
// second node definition

// possibly further node blocks

#LINKS
// link definitions
```

In Sections 3.1 to 3.3, we discuss the different blocks and their associated syntax rules in turn. However, we first discuss the basic elements that are common to all blocks.

Comments

Note that in the above code excerpt, the contents of the various blocks are omitted in favor of end-of-line comments. Such comments are initiated by a double forward slash (`//`) and terminated by a line feed and their content will be ignored.

Identifiers

Note furthermore that the nodes in the code excerpt are assigned identifiers. Identifiers are used to name different kinds of language objects such as nodes and variables. In all cases, identifiers may contain *letters*, *numbers*, *underscores*, and *dollar signs* but must begin with a *letter* or an *underscore*. Accordingly, the following are valid identifiers.

`mynode1`, `_SolarPlant_2`, `HydroStorage_$`

Beyond the lexical requirements, identifiers must also be unique in their respective scope. As such, no two nodes may have the same identifier since this would prohibit the unambiguous identification of a particular node. Similarly, variables and parameters may not have the same identifier as a node or other variables or parameters belonging to the same node. However, the same identifier may be reused to define variables or parameters that belong to different nodes.

Numbers

In terms of numbers, GBOML recognizes *floating-point numbers* and *integers*. Depending on the context, an *integer* may be called for, but in contexts where a *floating-point number* is required, an *integer* will also be accepted and converted to a *floating-point number*.

Expressions

Mathematical expressions are used to define the various constraint and objective functions involved in the problem model discussed in Section 2. Furthermore, numbers and identifiers referring to parameters and variables are used to construct such mathematical expressions. While numbers are always scalar, variables are always vectors with their entries being accessed via an index in brackets (`[` and `]`), and parameters may be either scalars or vectors. Indeed, letting `v` be an identifier referring to a vector quantity, the entries of `v` are accessed via

`v[0]`, `v[1]`, `v[2]`, ...

Note that the first index is `0` rather than `1`.

GBOML provides the following operators for elementary floating-point arithmetic, which are listed in order of decreasing precedence:

- Exponentiation (`**`)
- Multiplication (`*`)
- Division (`/`)
- Addition (`+`)
- Subtraction (`-`)

Operator precedence can be overridden by using parentheses (`(` and `)`). Beyond these elementary operators, GBOML provides the modulo operator as a native function:

- Modulo (`mod(<dividend>,<divisor>)`)

Given the above rules and letting `x` and `v` be a scalar and a vector, respectively, the following are valid GBOML expressions.

```
x**2 - 6.5*x - 9,     2*(x - 2.5)**3,     v[0] + v[mod(3,2)]
```

Note that the above expressions contain whitespace characters, which are not required. Indeed, all kinds of whitespace characters (space, tabulation, line feed, form feed, and carriage return) are ignored by the GBOML compiler.

Conditions

Besides their immediate use in model equations, expressions can be further used to construct logical conditions. Recall from (1) and (2) that the problem model discussed in Section 2 permits the expansion of constraints over a subset $\mathcal{T}_k$ of the set of time periods. These subsets are modeled by logical conditions that determine which time periods are to be included. The logical conditions in turn are constructed from expressions, comparison operators, and logical operators. The available comparison operators are the following:

- Equal (`==`)
- Not equal (`!=`)
- Less or equal (`<=`)
- Greater or equal (`>=`)
- Less (`<`)
- Greater (`>`)

The following are the available logical operators (in order of decreasing precedence):

- Negation (`not`)
- Conjunction (`and`)
- Disjunction (`or`)

As is the case with the algebraic operators discussed above, the precedence of logical operators can be overridden by using parentheses (`(` and `)`).

Given these operators and letting `t` denote an integer, the following are valid GBOML conditions.

```
t > 0 and t <= 10,     t < 2 or t > 4,     not mod(t,5) == 0
```

For the use of conditions in selectively enforced constraints, refer to Section 3.2.3.

Table 1 lists all algebraic and logical operators along with their precedence and associativity.

Table 1: Operator precedence and associativity

Precedence	Operator	Associativity
1	`**`	Left
2	`-` (unary)	Right
3	`*` `/`	Left
4	`+` `-` (binary)	Left

5	`==`
	`!=`
	`<=`
	`>=`
	`<`
	`>`
		None

6	`not`	Right
7	`and`	Left
8	`or`	Left

3.1 The `#TIMEHORIZON` Block

The `#TIMEHORIZON` block contains global information about the time indexing of variables throughout the model. In particular, a time horizon is defined as follows.

```
#TIMEHORIZON
T = <expression>;
```

Therein, `<expression>` is a mathematical expression that should evaluate to a positive integer. If `<expression>` evaluates to a positive but non-integral value, it will be rounded to the closest integer and a warning will be emitted. Expressions that cannot be evaluated and expressions that evaluate to a negative value are not permitted. In particular, since the `#TIMEHORIZON` block is the first block of any input file and no parameters have been defined before it, `<expression>` may not depend on any parameters. An example of a valid `#TIMEHORIZON` block is given below.

```
#TIMEHORIZON
T = 5.5*24+6;
```

The definition of a time horizon has two effects on the remainder of the model. First, the time horizon can be addressed as a parameter in the remainder of the model by referring to its identifier `T`. Accordingly, the identifier `T` is reserved and cannot be reused for the definition of nodes, variables, or parameters in the remainder of the model. Second, all variables defined in the model are indexed with respect to time, meaning that every variable is a vector with one entry for each time step in $\{0, 1, \ldots, \mathtt{T}-1\}$ (which corresponds to $\mathcal{T}$ in Section 2). Let, for example, `x` be a model variable. Then, the entries of `x` at the various time steps are accessed via

`x[0]`, `x[1]`, ... `x[T-1]`.

Furthermore, there are constraints that expand over the time horizon and use the identifier `t` as a variable time index for this expansion. Accordingly, `t` is a reserved identifier that cannot be used for any other purpose. For additional information on this kind of constraint, refer to Section 3.2.3.

3.2 The #NODE Block

Each #NODE block is associated with a unique identifier and is further divided into code blocks for the definition of parameters, variables, constraints, and objectives, respectively. Each of these blocks is introduced by one of the keywords #PARAMETERS, #VARIABLES, #CONSTRAINTS, and #OBJECTIVES. Then, a typical #NODE block is structured as follows.

```
#NODE <node identifier>
#PARAMETERS
// parameter definitions

#VARIABLES
// variable definitions

#CONSTRAINTS
// constraint definitions

#OBJECTIVES
// objective definitions
```

3.2.1 Parameter Definitions

A parameter definition maps an identifier to a fixed value, which may be either a scalar or a vector. The identifier must be unique within a given #NODE block and a value can be assigned to a parameter through one of the following three syntax rules.

```
<identifier> = <expression>;
<identifier> = {<term>,<term>,...};
<identifier> = import "<filename>";
```

First, a scalar parameter is defined according to the first rule. Therein, <expression> may be a scalar expression that evaluates to any floating-point number. In particular, it may contain parameter values that have been defined previously. If the parameter definition is valid with respect to these rules, a parameter named <identifier> will be created and assigned the value of <expression>.

Second, a vector parameter can be defined directly by providing a comma-separated list of values according to the second rule. Therein, each <term> may be a floating-point number, a previously defined scalar parameter, or an entry of a previously defined vector parameter. The resulting vector parameter can be indexed in order to retrieve its constituent entries.

Third, a vector parameter can be defined by providing an input file according to the third rule. Therein, <filename> refers to an input file in one of several delimiter-separated formats. The supported delimiter characters are *comma*, *semicolon*, *space*, and *line feed*. In contrast to the direct way of defining a vector parameter, the input file may only contain floating-point values and may not refer to other parameters.

Given these syntax rules, the following is an example of a valid #PARAMETERS block.

```
#PARAMETERS
pi     = 3.1416;
two_pi = 2*pi;
data   = import "data.csv";
angles = {0,data[2],two_pi};
```

Notably, the example makes use of the fact that previously-defined parameters can be used for the definition of new parameters. However, recall that parameter definitions are local to the present node and parameters defined in different nodes cannot be referred to in this context. The only exception to this rule is the time horizon T which is defined globally and is therefore not specific to any node.

3.2.2 Variable Definitions

Similar to parameter definitions, variable definitions are local to a given node. However, in contrast to parameters, variables are declared according to the following syntax rules with one of the keywords `internal`, `input`, and `output`.

```
internal : <identifier>;
input    : <identifier>;
output   : <identifier>;
```

While `internal` variables are meant to model the internal state of a node, `input` and `output` variables are meant to model the interaction between different nodes. That is, the interaction between two nodes is modeled by imposing constraints on the `input` and `output` variables of these nodes. For additional information on the modeling of node interactions, refer to Section 3.3.

Given these syntax rules, the following is an example of a valid `#VARIABLES` block.

```
#VARIABLES
internal : x;
internal : y;
input    : inflow;
output   : outflow;
```

In this example, the variables `x` and `y` are used to model the internal state of the node, while `inflow` and `outflow` are used to model the interaction with other nodes. Furthermore, as mentioned previously, all variables are automatically indexed with respect to time.

3.2.3 Constraint Definitions

Given the definitions of parameters and variables in the previous sections, the behavior of a node can be modeled by imposing constraints on its variables. The following are the syntax rules for the definition of basic equality and inequality constraints.

```
<expression> == <expression>;
<expression> <= <expression>;
<expression> >= <expression>;
```

Therein, both the left-hand side and the right-hand side of the constraints are general expressions while the type of constraint is indicated by the appropriate operator. As is common in linear programming, strict inequalities are not permitted in the definition of constraints since these generally entail problems with nonclosed feasible sets. Furthermore, and in line with the fact that parameter and variable definitions are local to a given node, the constraints must not reference quantities that are defined in other nodes.

Given these syntax rules, the following is an example of valid constraint definitions within an appropriate node and time horizon context.

```
#TIMEHORIZON
T = 2;

#NODE mynode

#PARAMETERS
a = {2,4};

#VARIABLES
internal : x;
output   : outflow;

#CONSTRAINTS
```

```
x[0] >= 0;
x[1] >= 0;
x[t] <= a[t];
outflow[1] == x[0] + x[1];

#OBJECTIVES
// objective definitions
```

Note that the variables and the parameter are only accessed at indexes that are valid according to the time horizon and the parameter's definition, respectively. Note also that the reserved time step identifier `t` is used on line 16 of the example. This results in the constraint being applied for all `t` in the time horizon. However, this expansion over the time horizon is predicated on the condition that the resulting constraint instances must be valid and invalid instances of the constraint are omitted. For example, the constraint

```
x[t] >= x[t-5];
```

will only be applied for values of `t` that lie in $[5, \mathtt{T} - 1]$ since the right-hand side of the constraint is ill-defined for $\mathtt{t} < 5$.

Recall that the problem formulation in (3) also permits the expansion of constraints over arbitrary subsets of the time horizon. In order to facilitate precise control over the expansion, GBOML provides the keywords `for` and `where`, which are used according to the following syntax rules.

```
<constraint> for t in [<start>:<end>];
<constraint> for t in [<start>:<step>:<end>];
<constraint> where <condition>;
<constraint> for t in [<start>:<end>] where <condition>;
<constraint> for t in [<start>:<step>:<end>] where <condition>;
```

Therein `<constraint>` is a basic equality or inequality constraint as defined previously. The first rule defines a constraint that is applied for all integral values of `t` that lie in the range between the `<start>` and `<end>`. Note that `<end>` must not be smaller than `<start>` for the range to be nonempty. If an empty range is given, a warning will be emitted. Furthermore, if `<end>` exceeds the time horizon, a warning will be emitted. The second rule makes use of the optional definition of a `<step>` that is used to increment through the range between `<start>` and `<end>`. The third rule defines a constraint that is applied for all values of `t` that lie in the time horizon and satisfy a logical condition (`<condition>`). Recall that such conditions are defined in terms of expressions, comparison operators, and logical operators. For a condition to be valid, it must be possible to evaluate it for a given `t`. In particular, conditions may depend on `t` and parameters but must not depend on variables. Finally, note in the fourth and fifth rule that the keywords `for` and `where` may also be combined in order to define a constraint that is applied for values of `t` that belong to a given range and satisfy a logical condition.

The following is an example that makes use of the keywords `for` and `where` in order to compactly write selectively-imposed constraints.

```
#TIMEHORIZON
T = 20;

#NODE mynode

#PARAMETERS
a = import "data.csv"; // parameter vector with 20 entries

#VARIABLES
internal : x;
output   : outflow;

#CONSTRAINTS
```

```
x[t] >= 0;
x[t] <= a[t] for t in [1:T-2];
x[t] == 0 where t == 0 or t == T-1;
outflow[0] == x[0];
outflow[t] == outflow[t-1] + x[t];

#OBJECTIVES
// objective definitions
```

While the syntax discussed above is sufficiently expressive to define general nonlinear constraints, the GBOML compiler expects the constraints to be affine with respect to all variables. This is in line with the fact that the problem model in Section 2 only permits linear programs.

3.2.4 Objective Definitions

Similar to the definition of constraints, objective definitions are given by a general expression and a keyword indicating whether the objective is to be minimized or maximized. The syntax rules for the definition of objectives are as follows.

```
min : <expression>;
max : <expression>;
```

At least one node in a given model must possess at least one objective but all nodes may have multiple objectives. In case multiple objectives are given, all objectives are aggregated into a single one by addition (respecting the sign associated with the keywords `min` and `max`). In the sense of the problem model in Section 2 where only a minimization is considered, this means that the signs of objectives to be maximized are inverted before summation. Furthermore, if the objective (i.e., if any of its terms or expressions) depends on the time index `t`, the expressions are expanded and summed over the time horizon. Overall, this aggregation corresponds to the inner sum in the objective function of the problem formulation in (3).

Extending the previous example by an objective definition, the following is an example of a complete and valid `#NODE` block in an appropriate time horizon context.

```
#TIMEHORIZON
T = 20;

#NODE mynode

#PARAMETERS
a = import "data.csv"; // parameter vector with 20 entries

#VARIABLES
internal : x;
output   : outflow;

#CONSTRAINTS
x[t] >= 0;
x[t] <= a[t] for t in [1:T-2];
x[t] == 0 where t == 0 or t == T-1;
outflow[0] == x[0];
outflow[t] == outflow[t-1] + x[t];

#OBJECTIVES
max : outflow[T-1];
```

As for constraint definitions, the syntax for objective definitions is sufficiently expressive to define nonlinear objectives. However, the GBOML compiler expects all objectives to be affine with respect to all variables in order to conform to the linearity requirements of the problem model in Section 2.

3.3 The `#LINKS` Block

The `#LINKS` block contains definitions of links between nodes, which represent the graph structure of the model. As mentioned previously, variable definitions are local to their respective nodes and a distinction is made between `internal`, `input`, and `output` variables. While general constraints can be defined for all variables belonging to a particular node (cf. Section 3.2.3), links can be defined for `input` and `output` variables across all nodes of the model. Furthermore, links may only express equality of variables rather than general constraints. The syntax for defining links is given as follows.

```
<output variable> == <input variable>,...,<input variable>;
```

Note that the syntax permits one to link one output variable to an arbitrary number of input variables by equality. Furthermore, all variables are identified by the appropriate `<node identifier>` and `<variable identifier>` according to

`<node identifier>.<variable identifier>`

Given these syntax rules, the following is an example of a valid `#LINKS` block.

```
#TIMEHORIZON
// time horizon definition

#NODE node1
#VARIABLES
output : x;
// further node content

#NODE node2
#VARIABLES
input : y;
// further node content

#NODE node3
#VARIABLES
input : z;
// further node content

#LINKS
node1.x == node2.y, node3.z;
```

3.4 Useful Idioms

With the GBOML syntax rules described above, there are several modeling needs that can be addressed by using particular idioms that may not be immediately apparent. Here, we discuss those modeling needs and propose the appropriate idioms to address them.

Scalar variables

As mentioned previously, all variables are vectors with one entry for each time step in the time horizon. If, however, a variable is meant to have only one value across the entire time horizon, this can be achieved by adding an appropriate constraint. Indeed, letting `x` be a variable, the constraint

```
x[t] == x[0];
```

ensures that `x` has the same value at each time step.

Repeating data

In some cases, the modeling task calls for repeating data. For example, a model may cover a time horizon of multiple days but the behavior of certain model components may be the same for each day. This case is illustrated in the following example along with the appropriate idiom for encoding the repeating model behavior.

```
#TIMEHORIZON
T = 5*24; // 5 days with an hourly resolution

#NODE factory

#PARAMETERS
production = import "production.csv"; // 24 values for hourly production

#VARIABLES
output : outflow;

#CONSTRAINTS
outflow[t] == production[mod(t,24)];

#OBJECTIVES
// objective definitions
```

4 Examples

In this section, we study two examples of optimization problems that can be easily modelled in the GBOML grammar. The first example deals with the installation of a microgrid system. We discuss the structure of the microgrid problem and show that it can be naturally encoded in the GBOML grammar presented in Section 3. The second example focuses on the modelling and design of remote carbon-neutral fuel supply chains [1]. The structure of such systems and how they may be represented in a graph-based modelling framework is discussed. For both examples, numerical results are reported and discussed.

4.1 Microgrid Example

A grid-connected microgrid is a small-scale and (ideally) self-sufficient electric power system. It consists of an interconnection of electric generators, e.g. solar panels or fossil fuel generators, and loads, namely the set of electricity consumers. An electrical storage system is often added to it in order to be able to balance the production and consumption of electricity without depending on the distribution network.

In this section, we develop a GBOML model to solve a sizing problem of an electric microgrid composed of solar panels, a battery and some consumers. The aim of the sizing problem is to determine the optimal capacities of the solar panels and battery storage system such that the costs generated by the installation and operation of the grid over twenty years is minimized. The electricity consumed in the microgrid and the solar irradiation of the panels are assumed known for a typical representative day.

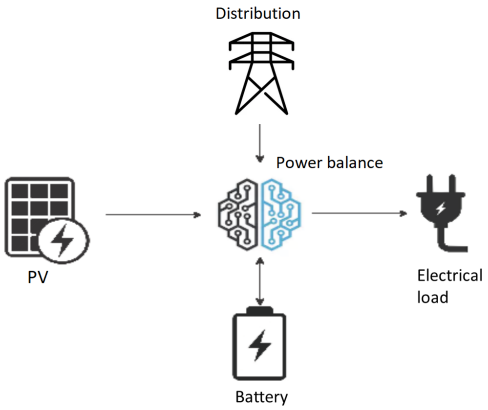


Figure 1: Microgrid configuration.

The model of the microgrid we propose to study is introduced in [2]. A graphical representation is provided in Figure 1. The system is composed of five nodes. The first node corresponds to solar panels. This node outputs power generated from sunlight. The second node represents the battery that takes as input a power flow that charges it. As output, the battery can also provide power when it discharges. The third node represents the consumer load. This node takes as input the power that is consumed by the users of the microgrid. The fourth node represents the electricity distribution network and can output power which is bought from this network. Finally, one node represents the interconnection and the power balance in the microgrid. This node is connected to the inputs and outputs of the other nodes and balances the power production and consumption by making up for any power shortage in the microgrid importing power from the distribution network. In case there is a power surplus due to a large production of solar energy, the surplus is curtailed and lost. Due to its particular configuration, this microgrid is an ideal candidate to demonstrate the strength of the GBOML language for modelling physical systems. A skeleton of the optimization problem in the language is provided in Listing 1. First, the horizon T is defined as the number of hours over the lifetime of the system, which is assumed to be twenty years. Then, the five nodes shall be implemented. Finally, these elements are linked to one another.

```
// The horizon corresponds to the number of hours in twenty years
#TIMEHORIZON
T = 20 * 365 * 24;

// Draft implementation of solar panel node
#NODE SOLAR_PV
#VARIABLES
output: electricity;

// Draft implementation of battery node
#NODE BATTERY
#VARIABLES
input: charge;
output: discharge;

// Draft implementation of demand node
#NODE DEMAND
#VARIABLES
input: consumption;

// Draft implementation of distribution network node
#NODE DISTRIBUTION
#VARIABLES
output: power_distribution;

// Draft implementation of power balance node
#NODE POWER_BALANCE
#VARIABLES
input: pv_production;
input: battery_discharge;
input: power_distribution;
output: battery_charge;
output: consumption;

#LINKS
SOLAR_PV.electricity == POWER_BALANCE.pv_production;
BATTERY.discharge == POWER_BALANCE.battery_discharge;
POWER_BALANCE.consumption == DEMAND.consumption;
POWER_BALANCE.battery_charge == BATTERY.charge;
DISTRIBUTION.power_distribution == POWER_BALANCE.power_distribution;
```

Listing 1: Skeleton of the code.

Let us now discuss the implementation of the five nodes one by one.

Solar panels node. A solar panel is a technology that produces energy from solar irradiance. Formally, the maximal quantity of power the panels can produce is referred to as the installed capacity of solar panels, which is denoted by $\bar{P}^{PV} \in \mathbb{R}^+$ (in watt). This capacity is a property of the material which is installed and is associated to a marginal cost π^{PV} (in currency/watt). The investment cost is thus the following:

$$I^{PV} = \bar{P}^{PV} \cdot \pi^{PV} \ . \tag{4}$$

At time t, the maximum power that may be generated by the solar panels is equal to the product of the installed capacity $\bar{P}^{PV}$ and a dimensionless capacity factor value $p_t^{PV} \in [0, 1]$, which is computed based on the irradiance at time t and the PV technology at hand. In addition, the power can be curtailed so that the actual power production $P_t^{PV} \in \mathbb{R}^+$ (in watt) can be smaller than the maximum power that may be generated by the panels. The latter is captured by the following inequality constraint between the power injected in the microgrid P_t^{PV} and the maximal power generated by the solar panels, the equality being reached when no curtailment occurs:

$$P_t^{PV} \leq p_t^{PV} \cdot \bar{P}^{PV} \ . \tag{5}$$

The node accounting for the solar panels is implemented in Listing 2. This node implements two internal variables: one for the capacity $\bar{P}^{PV}$ and one for the investment cost I^{PV}. Its output is the power generated by the panels P_t^{PV}. The constraint (5) on these variables is also implemented in this node. Finally, the objective to be minimized is the investment cost I^{PV} from equation (4). Let us note that the capacity $\bar{P}^{PV}$ shall be constant across the full optimization horizon. The latter is captured by using the idiom for scalar variables (cf. Section 3.4) on line 11 in Listing 2. Furthermore, the idiom for repeating data is used on line 14 in Listing 2 in order to repeat the solar irradiance time series for each day in the time horizon.

```
1  #NODE SOLAR_PV
2  #PARAMETERS
3  pi = 1;
4  irradiance_time_series = import "pv_gen.csv";
5  max_capacity = 1000;
6  #VARIABLES
7  internal: capacity;
8  internal: investment;
9  output: electricity;
10 #CONSTRAINTS
11 capacity[t] == capacity[0];
12 capacity[t] >= 0;
13 capacity[t] <= max_capacity;
14 electricity[t] <= irradiance_time_series[mod(t, 24)] * capacity[t];
15 investment[t] == (capacity[t] * pi) / T;
16 #OBJECTIVES
17 min: investment[t];
```

Listing 2: Solar PV node.

Battery node. A battery is an electrical device that can store energy up to a quantity $\bar{E}^B \in \mathbb{R}^+$ (in watt-hour) called installed capacity. Similarly to the solar panels, a marginal cost π^B (in currency/watt-hour) is associated with the deployment of units of installed capacity, such that the total investment

cost is computed as follows:

$$I^B = \bar{E}^B \cdot \pi^B \ . \tag{6}$$

Energy can be charged or discharged from the battery by letting power flow in or out of the battery. The charging power and discharging power are denoted by $P_t^{B+} \in \mathbb{R}^+$ (in watt) and $P_t^{B-} \in \mathbb{R}^+$ (in watt), respectively. The state of charge of the battery refers to the energy stored in the battery, it is denoted by $SoC_t^B \in \left[0, \bar{E}^B\right]$ (in watt-hour), and it is linked to the previous variables through the following constraint:

$$SoC_{t+1}^B = SoC_t^B + \eta \cdot P_t^{B+} - \frac{P_t^{B-}}{\eta} \ , \tag{7}$$

where $\eta \in [0, 1]$ is the efficiency of the battery, a physical parameter of the device quantifying the energy lost when charging and discharging the battery.

In addition to two internal variables representing the installed capacity $\bar{E}^B$ and the investment costs I^B, the charge P_t^{B+} and discharge P_t^{B-} of the battery are provided as the input and the output of the battery node, respectively. The investment cost I^B from equation (6) is minimized. The implementation is provided in Listing 3. In the latter, the constraint on line 13 is added to ensure that $\bar{E}^B$ is constant over the entire optimization horizon.

```
1  #NODE BATTERY
2  #PARAMETERS
3  eta = 0.75;
4  pi = 1;
5  max_capacity = 1000;
6  #VARIABLES
7  internal: capacity;
8  internal: investment;
9  internal: soc;
10 input: charge;
11 output: discharge;
12 #CONSTRAINTS
13 capacity[t] == capacity[0];
14 capacity[t] >= 0;
15 capacity[t] <= max_capacity;
16 soc[t] >= 0;
17 soc[t] <= capacity[t];
18 charge >= 0;
19 discharge >= 0;
20 soc[t+1] == soc[t] + eta * charge[t] - discharge[t] / eta;
21 investment[t] == (capacity[t] * pi / T);
22 #OBJECTIVES
23 min: investment[t];
```

Listing 3: Battery node.

Demand node. In the node shown in Listing 4, the electrical consumption, which is denoted by $P_t^C \in \mathbb{R}^+$ (in watt), is computed for each time t. This computation relies on a time series provided as parameter that gives the typical consumption for the 24 hours of a representative day.

```
#NODE DEMAND
#PARAMETERS
demand = import "demand.csv";
#VARIABLES
input: consumption;
#CONSTRAINTS
consumption[t] == demand[mod(t, 24)];
```

Listing 4: Demand node.

Distribution node. The distribution node represents the distribution network to which the microgrid is connected. It is possible to buy any quantity of power $P_t^{distribution} \in \mathbb{R}^+$ (in watt) on this grid to supply potential power shortages in the microgrid at time t. This power is bought at a marginal price $\pi^{distribution}$ (in currency/watt) such that the operational cost at time t is given by:

$$o_t = P_t^{distribution} \cdot \pi^{distribution} \tag{8}$$

In the current node, the total operational cost over the lifetime of the equipment is minimized. The objective to be minimized is thus the following:

$$O = \sum_{t=0}^{T-1} o_t \ . \tag{9}$$

This node is implemented in Listing 5. The node outputs $P_t^{distribution}$ and the operational cost o_t is computed in an internal variable. Finally, the total operational cost O from equation (9) is minimized.

```
#NODE DISTRIBUTION
#PARAMETERS
pi = 1;
#VARIABLES
internal: operational;
output: power_distribution;
#CONSTRAINTS
power_distribution[t] >= 0;
operational[t] == power_distribution[t] * pi;
#OBJECTIVES
min: operational[t];
```

Listing 5: Distribution node.

Power balance node. Finally, the node implemented in Listing 6 represents the operation of the microgrid. This node balances the production and consumption at any time and buys the power shortage in the microgrid from the electrical distribution operator. The solar production P_t^{PV} and the power discharged from the battery P_t^{B-} are provided as input. The power bought from the distribution network $P_t^{distribution}$ is also provided as input to the node. The node outputs the power charged in the battery P_t^{B+} and the power consumed P_t^{C}. The following constraint accounts for the power balance in the microgrid:

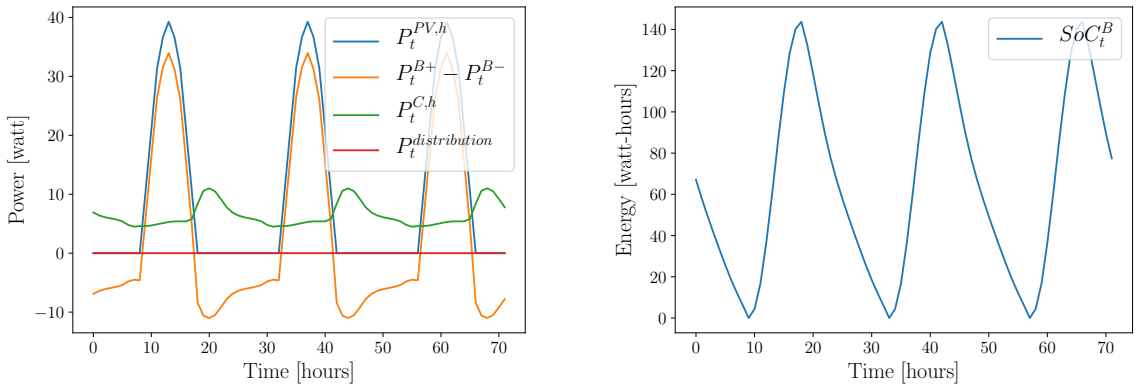
$$P_t^{PV} + P_t^{B-} + P_t^{distribution} = P_t^{B+} + P_t^{C} \ . \tag{10}$$

```
#NODE POWER_BALANCE
#PARAMETERS
pi = 1;
#VARIABLES
input: pv_production;
input: battery_discharge;
input: power_distribution;
output: battery_charge;
output: consumption;
#CONSTRAINTS
pv_production[t] + battery_discharge[t] + power_distribution[t] ==
    battery_charge[t] + consumption[t];
```

Listing 6: Power balance node.

Finally, the complete model is obtained by substituting the node implementations (Listings 2 to 6) in the skeleton code (Listing 1). The model is then translated using the GBOML compiler and solved with the *Gurobi* solver. A capacity $\bar{P}^{PV} = 261.8W$ of solar panels is installed and a battery with capacity $\bar{E}^{B} = 143.7Wh$ is selected. Figure 2a and Figure 2b represent the power balance and the state of charge of the battery, respectively, during three successive days (the first day of the optimization horizon is not shown as it is subject to border/transient effects). The sum of the three curves in Figure 2a is equal to zero to satisfy equation (10). Let us first note that the solar production peaks during midday and is equal to zero at night. The battery is then charged during this peak and discharged during the night to supply the demand. This strategy allows the microgrid not to buy electricity from the distribution network and it is therefore self-sufficient. The latter strategy results from the low investment costs for the solar panels and the battery compared with the electricity cost on the grid. For this optimal configuration, the objective function is such that:

$$\min \sum_{t=1}^{T} \Big(\underbrace{P_t^{distribution} \cdot \pi^{distribution}}_{\text{Operation}} + \underbrace{\bar{P}^{PV} \cdot \frac{\pi^{PV}}{T}}_{\text{Investment PV}} + \underbrace{\bar{P}^{B} \cdot \frac{\pi^{B}}{T}}_{\text{Investment battery}} \Big) = 405.5\,. \tag{11}$$



(a) Power balance in the microgrid.

(b) State of charge of the battery.

4.2 Remote Carbon-Neutral Fuel Supply Chains

The production of carbon-neutral synthetic fuels in remote areas where high-quality renewable resources are abundant has long been viewed as a means of developing a cost-effective, decarbonised energy supply

for countries with limited local renewable potential [4]. The synthesis of carbon-neutral fuels relies on a set of tightly-integrated technologies implementing various chemical processes. In order to properly estimate the cost of the final product, the entire supply chain must be modelled in an integrated fashion, from remote electricity production to product delivery at the destination.

From a modelling perspective, remote carbon-neutral fuel supply chains can be naturally represented as graphs where each node models a technology or process and has a low degree (i.e., each technology only interacts with a small subset of all technologies and processes), as depicted in Figure 3 for the particular case of synthetic methane. Moreover, for the purposes of strategic techno-economic analyses, each process shown in Figure 3 can be described using one of only three simple generic nodes, namely *conversion*, *storage* and *conservation* nodes [1]. Roughly speaking, conversion nodes represent technologies implementing physical processes that enable the transformation of commodities, storage nodes model technologies that can hold commodities over time and restore them when needed, and conservation nodes are used to enforce the conservation of flows of commodities between conversion and storage nodes. These nodes and their interactions are simple to encode in GBOML, which therefore provides a convenient way of building integrated carbon-neutral fuel supply chain models.

In [1], the GBOML modelling framework is presented and leveraged to study the economics of producing carbon-neutral synthetic methane from solar and wind energy in North Africa and exporting it to Northwestern Europe. The supply chain schematically represented in Figure 3 is modelled in an integrated fashion, and each technology in the supply chain is sized based on an operational horizon of one year with hourly resolution in order to minimize total system costs. The full supply chain model is described in detail in the paper, along with the data required to instantiate it. Results suggest that total system costs would be around 20B€ and 15B€ by 2030 for systems producing 100 TWh (higher heating value) of synthetic methane annually using solar-only plants and a combination of solar and wind power plants (assuming a weighted average cost of capital of 7%), respectively, resulting in carbon-neutral synthetic methane costs around 200€/MWh and 150€/MWh. If financing costs were neglected, results suggest that carbon-neutral synthetic methane costs would be around 87€/MWh and 124€/MWh for hybrid solar-wind and solar-only configurations, respectively. The input file encoding the model in GBOML and enabling the replication of these results is available at [8].

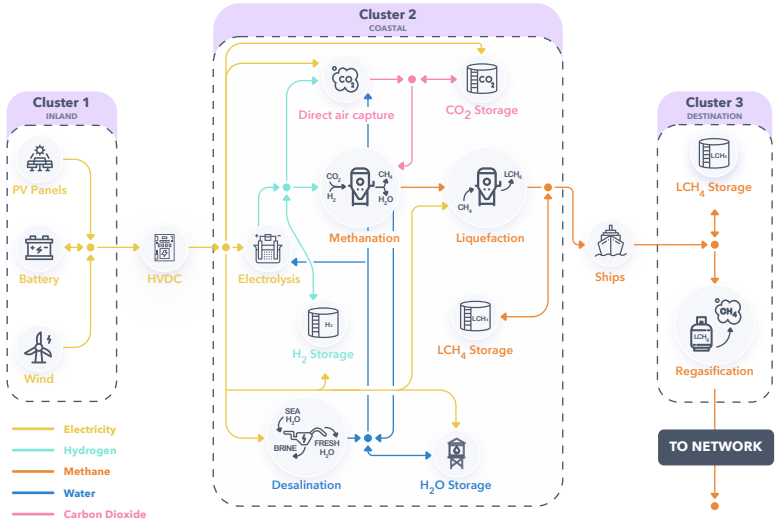


Figure 3: Schematic of remote carbon-neutral fuel supply chain [1]. Bullets represent conservation nodes, while icons represent conversion and storage nodes.

5 Conclusions

In this paper, we present the graph-based optimization modeling language (GBOML), which can be used to encode optimization problems. In contrast to popular and purely algebraic modeling language, GBOML is partially object-oriented and provides native facilities for encoding linear programs that have an underlying graph structure. Furthermore, GBOML provides a native way to implement models with discrete-time dynamics. A compiler for the language is available that interfaces with several linear programming solvers to solve the encoded problems.

First, we discuss the conceptual problem model that GBOML is built to capture. Second, we provide the syntax rules that make up the language and can be used to encode the graph structure of a model as well as the model equations. The abstract descriptions of the syntax rules are augmented by examples of valid GBOML expressions. Finally, we present two case studies that model optimization problems in energy systems using GBOML. The first example is concerned with the optimal sizing of renewable generation and battery storage in a micro grid. The second example is concerned with the optimal design and operation of a carbon-neutral fuel supply chain. In both cases, the features of GBOML are leveraged in order to model the optimization problem in a modular and concise way.

References

[1] Mathias Berger, David Radu, Ghislain Detienne, Thierry Deschuyteneer, Aurore Richel, and Damien Ernst. Remote Renewable Hubs for Synthetic Fuel Production. *http://hdl.handle.net/2268/250796*, 2020.

[2] Adrien Bolland, Ioannis Boukas, François Cornet, Mathias Berger, and Damien Ernst. Learning Optimal Environments using Projected Stochastic Gradient Ascent. *arXiv preprint arXiv:2006.01738*, 2020.

[3] Antonio J Conejo, Enrique Castillo, Roberto Mínguez, and Raquel García-Bertrand. Decomposition Techniques in Mathematical Programming Engineering and Science Applications. 2006.

[4] K Hashimoto, M Yamasaki, K Fujimura, T Matsui, K Izumiya, M Komori, A.A El-Moneim, E Akiyama, H Habazaki, N Kumagai, A Kawashima, and K Asami. Global CO2 Recycling: Novel Materials and Prospect for Prevention of Global Warming and Abundant Energy Supply. *Materials Science and Engineering: A*, 267(2):200 – 206, 1999. ISSN 0921-5093. doi:https://doi.org/10.1016/S0921-5093(99)00092-1.

[5] Josef Kallrath. *Algebraic Modeling Systems*. Springer, 2012.

[6] Kim Kibaek, Victor Zavala, Christian Tjandraatmadja, and Byeon Geunyeong. DSP Repository (Decomposition of Structured Programs), 2020. URL https://github.com/Argonne-National-Laboratory/DSP.

[7] Bardhyl Miftari. Graph-Based Optimization Modeling Language. Master's thesis, University of Liège, 2021.

[8] Bardhyl Miftari, Mathias Berger, Adrien Bolland, Hatim Djelassi, and Damien Ernst. GBOML Repository: Graph-Based Optimization Modeling Language, 2021. URL https://gitlab.uliege.be/smart_grids/public/gboml.

[9] Hermann Schichl. *Theoretical Concepts and Design of Modeling Languages for Mathematical Optimization*, pages 45–62. Springer US, Boston, MA, 2004.