Graph-Based Optimization Modeling Language: A Tutorial

Mathias Berger Adrien Bolland Bardhyl Miftari Hatim Djelassi
Damien Ernst

June 12, 2021

This paper introduces the graph-based optimization modeling language (GBOML), which enables the easy implementation of a broad class of structured mixed-integer 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 hypergraph. The language combines elements of 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 tools like 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 mixed-integer 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 hypergraph. GBOML draws on concepts used in 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 hyperedges represent the relationships between nodes through equality and inequality constraints. 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 mixed-integer linear programming solvers, namely Gurobi, CPLEX and Cbc.

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 hypergraph [1]. A graph abstraction is therefore employed to represent them, wherein nodes model optimization subproblems, while hyperedges express the relationships between nodes. A global discretized time horizon and associated set of time periods common to all nodes are also defined. Each node is equipped with a set of so-called *internal* and *external* (or *coupling*) variables. 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 hyperedge, constraints involving the coupling variables of the nodes to which it 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 $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a (possibly directed) hypergraph encoding the block structure of the problem at hand, with node set $\mathcal{N}$ and hyperedge set $\mathcal{E} \subseteq 2^{\mathcal{N}}$ (i.e., each hyperedge corresponds to a subset of nodes), let T be the time horizon considered and let $\mathcal{T} = \{0, 1, \ldots, T-1\}$ be the associated set of time periods. Let $X^n \in \mathcal{X}^n$ and $Z^n \in \mathcal{Z}^n$ denote the collection of internal and coupling variables defined at node $n \in \mathcal{N}$. Note that variables may take values in discrete or continuous sets. In addition, for any hyperedge $e \in \mathcal{E}$, let $Z^e = \{Z^n | n \in e\}$ denote the collection of coupling variables associated with each node to which this hyperedge is incident.

Let F^n denote the function defining the local objective at node $n \in \mathcal{N}$. In this paper, we consider scalar objectives of the form

$$F^n(X^n, Z^n) = f_0^n(X^n, Z^n) + \sum_{t \in \mathcal{T}} f^n(X^n, Z^n, t), \tag{1}$$

where f_0^n and f^n are (scalar) affine functions of X^n and Z^n.

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

$$h_k^n(X^n, Z^n, t) = 0, \ \forall t \in \mathcal{T}_k^n, \tag{2}$$

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

$$g_k^n(X^n, Z^n, t) \leq 0, \ \forall t \in \bar{\mathcal{T}}_k^n, \tag{3}$$

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

Likewise, both equality and inequality constraints may be defined over any hyperedge $e \in \mathcal{E}$. These constraints, however, can only involve the coupling variables of the nodes to which hyperedge $e \in \mathcal{E}$ is incident (i.e., nodes such that $n \in e$). More precisely, let H^e and G^e be affine functions of Z^e used to define the equality and inequality constraints associated with a given hyperedge $e \in \mathcal{E}$.

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

$$\begin{aligned} \min \quad & \textstyle\sum_{n \in \mathcal{N}} F^n(X^n, Z^n) \\ \text{s.t.} \quad & h_k^n(X^n, Z^n, t) = 0, \ \forall t \in \mathcal{T}_k^n, \ k = 1, \ldots K^n, \ \forall n \in \mathcal{N} \\ & g_k^n(X^n, Z^n, t) \leq 0, \ \forall t \in \bar{\mathcal{T}}_k^n, \ k = 1, \ldots \bar{K}^n, \ \forall n \in \mathcal{N} \\ & H^e(Z^e) = 0, \ \forall e \in \mathcal{E} \\ & G^e(Z^e) \leq 0, \ \forall e \in \mathcal{E} \\ & X^n \in \mathcal{X}^n, Z^n \in \mathcal{Z}^n, \ \forall n \in \mathcal{N}. \end{aligned} \tag{4}$$

Figure 1 schematically illustrates the class of problems that can be modelled in this framework.

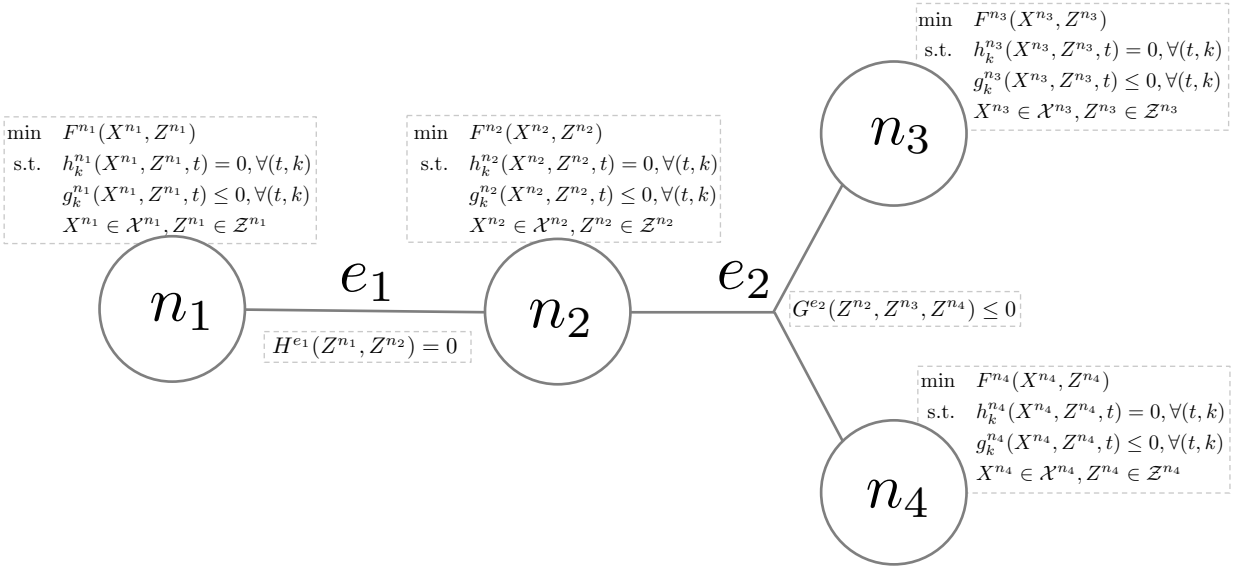


Figure 1: Graph abstraction of a hypothetical problem whose block structure is represented by four nodes (i.e., $\mathcal{N} = \{n_1, n_2, n_3, n_4\}$) and two hyperedges (i.e., $\mathcal{E} = \{e_1, e_2\}$, with $e_1 = \{n_1, n_2\}$ and $e_2 = \{n_2, n_3, n_4\}$). Note that e_1 only has equality constraints while e_2 only has inequality constraints in this example.

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. This input file is structured into blocks, which are introduced by one of the following keywords, namely `#TIMEHORIZON`, `#GLOBAL`, `#NODE`, and `#HYPEREDGE`. The time horizon information is given in the `#TIMEHORIZON` block, which must be the first one defined in the input file. Global parameters must be defined in the `#GLOBAL` block, which must come in second position. Then, each node can be defined in a `#NODE` block, while each

hyperedge can be defined in a `#HYPEREDGE` block. Note that the order in which `#NODE` and `#HYPEREDGE` blocks appear in the input file does not matter (i.e., hyperedge definitions may precede node definitions and vice-versa). Thus, an input file is typically structured as follows.

```
#TIMEHORIZON
// time horizon definition

#GLOBAL
// global parameters

#NODE <identifier>
// first node definition

#NODE <identifier>
// second node definition

#HYPEREDGE <identifier>
// first hyperedge definition

// possibly further node blocks

#HYPEREDGE <identifier>
// second hyperedge definition

// possibly further hyperedge blocks
```

In Sections 3.1 to 3.4, 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 by the compiler.

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

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*. It should be noted that scientific notation is supported and

automatically converted to *floating-point numbers*. Accordingly, the following are considered *floating-point numbers*,

```
1e-5,    -2.5e-10,    2e10.
```

Expressions

Mathematical expressions are used to define the various functions appearing in the constraints and objective of the abstract 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 and parameters may be either scalars or vectors of any length with their entries being accessed via an index in brackets (`[` and `]`). Indeed, letting `v` be an identifier referring to a vector quantity of length `L`, the entries of `v` are accessed via

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

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 and the sum operator as native functions:

- Modulo (`mod(<dividend>, <divisor>)`)
- Sum (`sum(<expression> for <id> in [<start>, <end>])`)

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,    sum((i - 2.5)/3 for i in [0:10]),    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 (2) and (3) that the abstract model discussed in Section 2 expands constraints over subsets $\mathcal{T}_k$ and $\bar{\mathcal{T}}_k$ of the set of time periods. These subsets are modeled by logical conditions that determine the time periods considered. The logical conditions 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 3.3.3.

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 the definition of the time horizon. The 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 raised. 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 can 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, constraints and objectives can be automatically expanded over the time full horizon by using the identifier `t` as a variable time index. In other words, the constraint or objectives that use `t` as an index are automatically expanded for each $\texttt{t} \in \{0, 1, ..., \texttt{T}-1\}$. 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 3.3.3.

3.2 The `#GLOBAL` Block

The non-mandatory `#GLOBAL` block contains the definition of parameters that can be accessed throughout the model. The global parameters are defined as follows,

```
#GLOBAL
// parameter definition
...
```

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 `#GLOBAL` 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 `#GLOBAL` block.

```
#GLOBAL
pi       = 3.1416;
two_pi   = 2*pi;
data     = import "data.csv";
len_data = 23;
angles   = {0,data[2],two_pi};
sum_data = sum(data[i] for i in [0:len_data-1]);
```

Notably, the example makes use of the fact that previously-defined parameters can be employed to define new parameters. Furthermore, the parameters defined in this block can be accessed in all the following blocks by referring to the parameters with the prefix `"global"`. In other words, all global parameters are referred to as,

`global.<parameter identifier>`

in the blocks that follow their definition.

3.3 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
```

3.3.1 Parameter Definitions

The parameters defined within a given `#NODE` block respect the same rules as those laid out in Section 3.2. However, the node parameters are local to the present node and parameters defined in different nodes cannot be referred to in this context.

For the sake of illustration, the following is a valid `#PARAMETERS` block.

```
#PARAMETERS
gravity       = 9.81;
speed         = import "speed.txt";
```

3.3.2 Variable Definitions

Variable definitions are also local to a given node. However, in contrast to parameters, variables are declared according to the following syntax rules with one of the two keywords `internal` and `external`. There exists two types of variables, namely scalar and vector variables.

```
internal : <identifier>;
external : <identifier>;
internal : <identifier> [ <expression> ];
external : <identifier> [ <expression> ];
```

Variables defined only by an identifier are scalar variables with the identifier giving the variable name (Rules 1 and 2). An expression can be added after the identifier to declare a vector variable and specify its length (Rules 3 and 4). While `internal` variables are meant to model the internal state of a node, `external` variables are meant to model the interaction between different nodes. That is, their coupling is modeled by imposing constraints on their `external` variables. For additional information on the modeling of node interactions, refer to Section 3.4.

Furthermore, variables can be `continuous`, `integer` or `binary`, where continuous variables are the default. Integral and binary variables are defined by adding the keywords `integer` and `binary` after the `internal/external` definition, respectively. The keyword `continuous` also exists but does not bring any new information.

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

```
#VARIABLES
internal integer : x; // internal integer scalar variable
internal binary  : y[T]; // internal binary variable vector of size T
external : inflow[1000]; // external continuous variable vector of size 1000
external : outflow[1000]; // external continuous variable vector of size 1000
```

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.

3.3.3 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 syntax rules for the definition of basic equality and inequality constraints are as follows

```
<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. 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[T];
external : outflow[T];

#CONSTRAINTS
x[0] >= 0;
x[1] >= 0;
x[2] <= a[3];
outflow[1] == sum(x[i] for i in [0:T-1]);

#OBJECTIVES
// objective definitions
```

Note that the variables and the parameter are only accessed at indices that are consistent their definitions.

Recall that the problem formulation in (4) also allows for the expansion of constraints over arbitrary subsets of the time horizon. GBOML provides two options to specify expansion ranges and define vectorized constraints, namely user-defined and automatic expansions.

First, user-defined expansions can be constructed as follows:

```
<constraint> <expansion range>;
```

where `<constraint>` is an equality or inequality constraint and `<expansion range>` can be expressed using the `for` and `where` keywords, according to the following syntax rules:

```
<expansion range> := for <identifier> in [<start>:<end>];
                  := for <identifier> in [<start>:<step>:<end>];
                  := for <identifier> in [<start>:<end>] where <condition>;
                  := for <identifier> in [<start>:<step>:<end>] where <condition>;
```

The first rule defines a constraint that is applied for all integral values of `<identifier>` that lie in the range between `<start>` and `<end>` (both included). Note that `<start>` must be smaller than `<end>` for the range to be nonempty. If an empty range is given, a warning will be raised. The `<identifier>` may be any identifier that has not been used to define a parameter or a variable in the present node block. The `t` identifier is reserved for automatic expansion (discussed below) and may not be used

for user-defined expansions. 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 and fourth rules are only extensions of the first two, where a certain `condition` needs to be satisfied for the constraint to be expanded. 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 value of `<identifier>`. In particular, conditions may depend on `<identifier>` and parameters but must not depend on variables. In addition, the indices over which expansions take place must be valid for vectors of parameters and variables involved in vectorized constraints. More precisely, an index is valid if it is non-negative and does not exceed the size of vector. If an index that is not valid is used in the expansion, an error is returned.

Second, automatic expansions can be declared by using the `t` identifier in a constraint. The constraint is then expanded over all valid indices $\texttt{t} \in \{0, ..., T-1\}$. For example, the following vectorized constraint:

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

will only be expanded over $\texttt{t} \in [5, \texttt{T}-1]$ since the right-hand side expression is ill-defined for $\texttt{t} < 5$. A warning is also raised to indicate the values of `t` over which the constraint cannot be expanded. Furthermore, a condition can be added in automatic expansions. The corresponding rule can be written as:

```
<constraint> <condition>;
```

where `condition` can depend on `t` and parameters.

The following is an example illustrating both expansion methods and making 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[T];
external : outflow[T];

#CONSTRAINTS
x[t] >= 0;
x[i] <= a[i] for i in [1:(T-2)/2];
0 <= a[i]*x[i] for i in [2:2:10] where i != 4;
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 constraints to be affine with respect to all variables, which is consistent with the structure of the abstract model discussed in Section 2. Hence, encoding nonlinear constraints leads to an error being raised.

3.3.4 Objective Definitions

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

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 in the same node block, all objectives are aggregated into a single one by summing them (respecting the sign associated with the keywords `min` and `max`). In the sense of the abstract model discussed in Section 2, which is a minimization problem, this means that the signs of objectives to be maximized are inverted before summation. Overall, this aggregation corresponds to the inner sum in the objective function of the problem formulation in (4). Objectives can also be expanded in two ways, namely via user-defined and automatic expansions. First, user-defined expansions make use of an `identifier` that will be expanded over each value in the `expansion range`. Second, automatic expansions can be constructed by using the `t` identifier directly in the objective. Note that due to all the objectives being aggregated, the following objectives are equivalent.

```
min : x[t],   min : sum(x[i] for i in [0:T-1])
```
(5)

The previous example can be completed by defining an objective function, which yields a complete and valid `#NODE` block.

```
#TIMEHORIZON
T = 20;

#NODE mynode

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

#VARIABLES
internal : x[T];
external : outflow[T];

#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.

3.4 The `#HYPEREDGE` Block

Each hyperedge is defined using a `#HYPEREDGE` block. A hyperedge must have an `<identifier>` and may have its own parameters. No two hyperedges may have the same identifier, while parameters defined in a hyperedge respect the exact same rules as the ones defined in `#NODE` blocks (cf. Section 3.3.1). A hyperedge typically couples variables belonging to different nodes via equality or inequality constraints (or both). More specifically, recall that a distinction is made between `internal` and `external` variables when defining them in `#NODE` blocks (cf. Section 3.3.2). While affine constraints involving all variables

declared in a `#NODE` block can be defined in the same block (cf. Section 3.3.3), constraints defined in `#HYPEREDGE` blocks couple `external` variables associated with any subset of nodes. The syntax for defining hyperedges is as follows.

```
#HYPEREDGE <identifier>
#PARAMETERS
// parameter definitions
#CONSTRAINTS
// constraint definitions
<expression> == <expression> <expansion range>;
<expression> <= <expression> <expansion range>;
```

The rules used to define constraints in `#HYPEREDGE` blocks are the same as the ones used for `#NODE` blocks (cf. Section 3.3.3). However, since external variables defined in different nodes may have the same local identifier, all variables must be identified by the appropriate `<node identifier>` and `<variable identifier>` when defining edges, that is,

`<node identifier>.<variable identifier>`

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

```
#TIMEHORIZON
// time horizon definition

#NODE node1
#VARIABLES
external : x;
external : inflow[T];
// further node content

#NODE node2
#VARIABLES
external : y;
external : outflow[T];
// further node content

#HYPEREDGE hyperedge2
#CONSTRAINTS
node1.inflow[t] == node2.outflow[t];

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

#HYPEREDGE hyperedge1
#PARAMETERS
weight = {1/3,2/3};
#CONSTRAINTS
node1.x <= weight[0]*node2.y + weight[1]*node3.z;
node2.y <= node3.z
```

3.5 Useful Idioms

Several modeling needs may be readily addressed by using particular idioms that may not be immediately apparent from the GBOML syntax rules described above. We discuss those modeling needs and propose the appropriate idioms to address them next.

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
external : outflow[T];

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

#OBJECTIVES
// objective definitions
```

Round down integer division

Integer division is not natively implemented in GBOML but can be a very useful tool for a lot of modeling tasks. To illustrate this, let us consider a model whose time horizon spans several days and some of whose components involve indices that correspond to hours and days, respectively. An example of such problems is given below with the appropriate idiom for encoding the different index behaviors.

```
#TIMEHORIZON
T = 365*24; // 365 days with hourly resolution

#GLOBAL
days = T/24;

#NODE bank

#PARAMETERS
interest_rate = import "interest_rate.csv"; // 365 values for daily interest
    rates
mean_interest = sum(interest_rate[i] for i in [0:global.days-1])/global.days;
    //mean interest rate

#VARIABLES
internal : investment_interest[T];
internal : investment[T];

#CONSTRAINTS
investment_interest[i] == interest_rate[(i-mod(i,24))/24]*investment[i] for i
    in [0:T-1];
```

4 Examples

In this section, two examples of optimization problems that can be easily modelled in the GBOML grammar are studied. The first example deals with the installation of a microgrid system. The structure of the microgrid problem is discussed and it is shown 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 (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 the problem of sizing 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.

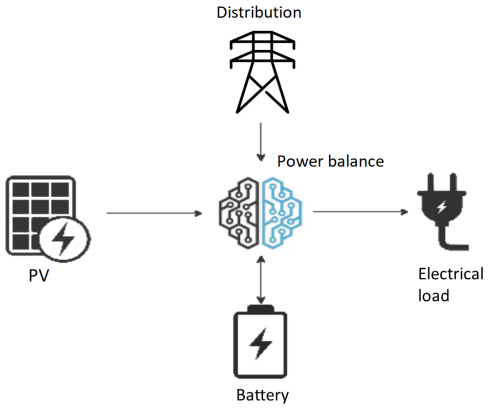


Figure 2: Microgrid configuration.

The model of the microgrid we propose to study is introduced in [2]. A graphical representation is provided in Figure 2. The system is composed of four nodes implementing the behavior of the elements of the microgrid. Additionally, their interactions are modeled using external variables that are associated with these nodes and using constraints on these variables modeled by hyperedges. The first node corresponds to solar panels. This node uses power generated from sunlight as an external variable. In case there is a power surplus due to a large production of solar energy, the surplus may be curtailed. The second node represents the dynamics and costs of a battery based on the power flows that charge and discharge it. The third node models the consumer load, which is also represented by an external variable. The last node represents the electricity distribution network and the power that is bought from the network is represented by an external variable. Finally, the external variables of these nodes are linked through a hyperedge representing the power balance in the system. A skeleton

of the optimization problem expressed 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 four nodes are implemented. Finally, these nodes 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
external: electricity[T];

// Draft implementation of battery node
#NODE BATTERY
#VARIABLES
external: charge[T];
external: discharge[T];

// Draft implementation of demand node
#NODE DEMAND
#VARIABLES
external: consumption[T];

// Draft implementation of distribution network node
#NODE DISTRIBUTION
#VARIABLES
external: power_distribution[T];

#HYPEREDGE POWER_BALANCE
// hyperedge modeling the power balance in the microgrid
```

Listing 1: Skeleton of the code.

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

Solar panels node. A solar panel is a technology that harnesses solar radiation for electricity production. 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). The installed capacity can be expanded at a marginal cost of π^{PV} (in currency/watt). The investment cost is thus the following:

$$I^{PV} = \bar{P}^{PV} \cdot \pi^{PV} \ . \tag{6}$$

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 parameter $p_t^{PV} \in [0, 1]$, 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:

$$P_t^{PV} \leq p_t^{PV} \cdot \bar{P}^{PV} \ . \tag{7}$$

The node accounting for the solar panels is implemented in Listing 2. This node has two scalar internal variables: one for the capacity $\bar{P}^{PV}$ and one for the investment cost I^{PV}. A time-dependent

external variable implements the power generated by the panels P_t^{PV}. Constraint (7) is also encoded in this node. Finally, the objective to be minimized is the investment cost I^{PV} defined in equation (6). Note that the idiom for repeating data (cf. Section 3.5) 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  external: electricity[T];
10 #CONSTRAINTS
11 capacity >= 0;
12 capacity <= max_capacity;
13 electricity[t] <= irradiance_time_series[mod(t, 24)] * capacity;
14 investment == capacity * pi;
15 #OBJECTIVES
16 min: investment;
```

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 a battery storage system, such that the total investment cost is computed as follows:

$$I^B = \bar{E}^B \cdot \pi^B \ . \tag{8}$$

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, which is denoted by $SoC_t^B \in \mathbb{R}^+$ (in watt-hour) and is upper-bounded by the installed capacity:

$$SoC_t^B \leq \bar{E}^B \ . \tag{9}$$

In addition, the state of charge is linked to the power flowing in and out of the battery through the following constraint:

$$SoC_{t+1}^B = SoC_t^B + \eta \cdot P_t^{B+} - \frac{P_t^{B-}}{\eta} \ , \tag{10}$$

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

In addition to two scalar internal variables representing the installed capacity $\bar{E}^B$ and the investment cost I^B, a time-dependent internal variable stands for the state of charge of the battery SoC_t^B. Furthermore, the charge P_t^{B+} and discharge P_t^{B-} of the battery are defined as time-dependent external variables of the battery node. Finally, the investment cost I^B from equation (8) is minimized. The implementation is provided in Listing 3. Note that a constraint has been added on line 17 in order to impose that the initial state of charge of the battery is equal to the final state of charge. This constraint is a common modeling trick used to avoid spurious transient effects in storage operation close to the beginning and the end of the time horizon.

```
#NODE BATTERY
#PARAMETERS
eta = 0.75;
pi = 1;
max_capacity = 1000;
#VARIABLES
internal: capacity;
internal: investment;
internal: soc[T];
external: charge[T];
external: discharge[T];
#CONSTRAINTS
capacity >= 0;
capacity <= max_capacity;
soc[t] >= 0;
soc[t] <= capacity;
soc[0] == soc[T-1];
charge[t] >= 0;
discharge[t] >= 0;
soc[t+1] == soc[t] + eta * charge[t] - discharge[t] / eta;
investment == capacity * pi;
#OBJECTIVES
min: investment;
```

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 based on a time series provided as a parameter giving the typical consumption for the 24 hours of a representative day.

```
#NODE DEMAND
#PARAMETERS
demand = import "demand.csv";
#VARIABLES
external: consumption[T];
#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 power $P_t^{distribution} \in \mathbb{R}^+$ (in watt) from this grid to supply any power shortage that may occur 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:

$$o_t = P_t^{distribution} \cdot \pi^{distribution} \tag{11}$$

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

$$O = \sum_{t=0}^{T-1} o_t \, . \tag{12}$$

This node is implemented in Listing 5. The power $P_t^{distribution}$ is modeled by a time-dependent external variable. Moreover, the operational cost o_t is computed using a time-dependent internal variable and the total operational cost O from equation (12) is minimized.

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

Listing 5: Distribution node.

Let us now discuss the link between the nodes, which is an equality constraint representing the balance between production and consumption of power in the microgrid. The sum of the solar production P_t^{PV}, the power discharged from the battery P_t^{B-} and the power bought from the distribution network $P_t^{distribution}$ must be equal to the sum of the power charged in the battery P_t^{B+} and the power consumed P_t^{C}. In other words, the following constraint accounts for the power balance in the microgrid:

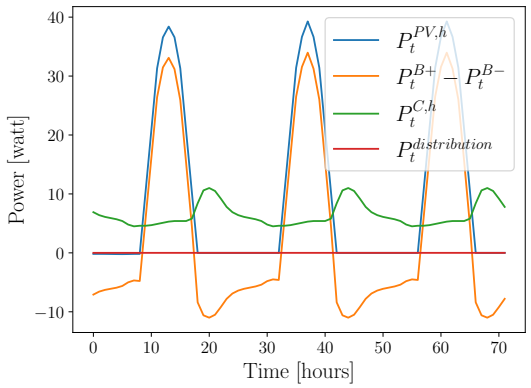
$$P_t^{PV} + P_t^{B-} + P_t^{distribution} = P_t^{B+} + P_t^{C} \ . \tag{13}$$

```
#HYPEREDGE POWER_BALANCE
#CONSTRAINTS
SOLAR_PV.electricity[t] + BATTERY.discharge[t] + DISTRIBUTION.
    power_distribution[t] == BATTERY.charge[t] + DEMAND.consumption[t];
```

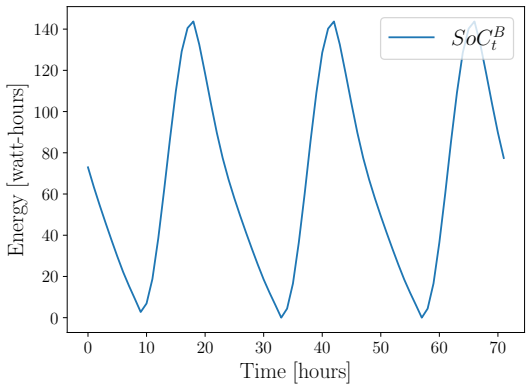
Listing 6: Power balance equality between nodes.

Finally, the complete model is obtained by substituting the 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. Figures 3a and 3b represent the power balance and the state of charge of the battery, respectively, during three successive days. The sum of the three curves in Figure 3a is equal to zero to satisfy equation (13). Note that the solar production peaks during midday and is equal to zero at night. The battery is thus 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 is a consequence of the low investment costs in the solar panels and battery storage system compared with the cost of purchasing electricity from the grid. For this optimal configuration, the objective function is such that:

$$\min \underbrace{\bar{P}^{PV} \cdot \pi^{PV}}_{\text{Investment PV}} + \underbrace{\bar{P}^{B} \cdot \pi^{B}}_{\text{Investment battery}} + \sum_{t=1}^{T} \underbrace{P_t^{distribution} \cdot \pi^{distribution}}_{\text{Operation}} = 405.5 \ . \tag{14}$$



(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 hypergraphs 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 4 for the particular case of synthetic methane. Moreover, for the purpose of strategic techno-economic analyses, each process shown in Figure 4 can be described using one of only two simple generic nodes, namely *conversion* and *storage* nodes [1]. Roughly speaking, conversion nodes represent technologies implementing physical processes that enable the transformation of commodities, while storage nodes model technologies that can hold commodities over time and restore them when needed. On the other hand, the relationship between nodes is expressed via so-called *conservation* hyperedges that enforce the conservation of flows of commodities between conversion and storage nodes. These nodes and hyperedges are simple to encode in GBOML, which therefore provides a convenient way of building integrated carbon-neutral fuel supply chain models.

In [1], the modelling framework is 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 4 is modelled in an integrated fashion, and each technology in the supply chain is sized based on an operational horizon of five years with hourly resolution in order to minimize total system cost. 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 1.5 B€/year (over the lifetime of the system) by 2030 for systems producing 10 TWh (higher heating value) of synthetic methane annually using 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 150 €/MWh. A comprehensive sensitivity analysis is also carried out in order to assess the impact of various techno-economic parameters and assumptions on synthetic methane cost, including the availability of wind power plants, the investment costs of electrolysis, methanation and direct air capture plants, their operational flexibility, the energy consumption of direct air capture plants, and financing costs. The most expensive configuration (around 200 €/MWh) relies on solar photovoltaic power plants alone, while the cheapest configuration (around 88 €/MWh) makes use of a combination of solar PV and wind power plants and is obtained when financing costs are set to zero. The input files encoding the model in GBOML and enabling the replication of these results are available

at [8].

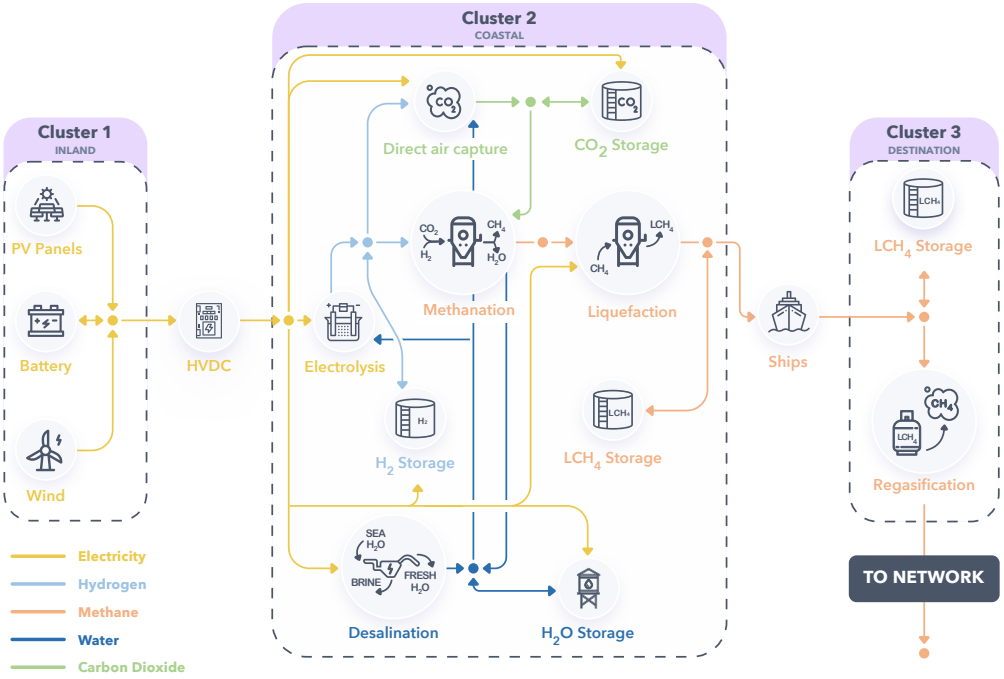


Figure 4: Remote carbon-neutral fuel supply chain [1]. Icons represent conversion and storage nodes, while bullets and arrows schematically represent hyperedges.

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 mixed-integer 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 mixed-integer linear programming solvers to solve the encoded problems.

First, we discuss the abstract problem class 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 model equations. The abstract description of syntax rules is 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 Carbon-Neutral Synthetic Fuel Production. *Frontiers in Energy Research*, 9:200, 2021. doi:10.3389/fenrg.2021.671279.

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