SMS++ File Format Reference

Contents

1 Introduction

This document describes the format of input files for the main components of the SMS++ project.

Insomuch as the set of types of optimization problems (Block in SMS++ parlance) to be solved within the project, and the set of available solution methods for doing it (Solver in SMS++ parlance) may be expanding over time, this document is intended as dynamic: as new components will added, the document will grow to cover the corresponding input formats. Non-backward-compatible modification of existing input formats will be kept to a minimum.

The self-documenting nature of netCDF, together with its portable readability and the schema definition of the various Block, will make it possible to re-use all or some blocks of one SMS++-generated model in other contexts.

2 General netCDF file structure

netCDF (network Common Data Form) [1] is a set of interfaces for array-oriented data access and a freely distributed collection of data access libraries for C, Fortran, C++, Java, and other languages. The netCDF libraries support a machine-independent format for representing scientific data. Together, the interfaces, libraries, and format support the creation, access, and sharing of scientific data.

netCDF data is:

- Self-Describing. A netCDF file includes information about the data it contains.
- Portable. A netCDF file can be accessed by computers with different ways of storing integers, characters, and floating-point numbers.
- Scalable. A small subset of a large dataset may be accessed efficiently.
- Appendable. Data may be appended to a properly structured netCDF file without copying the dataset or redefining its structure.
- Sharable. One writer and multiple readers may simultaneously access the same netCDF file.
- Archivable. Access to all earlier forms of netCDF data will be supported by current and future versions of the software.

For the purpose of this document, only a relatively small set of netCDF concepts need be discussed.

- netCDF *file*: is the basic container of netCDF, e.g. corresponding to a standard file in the filesystem of the host machine. In the C++ interface used by SMS++ it corresponds to an object of type `netCDF::NcFile`. A typical constructor of the object requires the pathname of the file (a standard `char *`) and the mode, which can be e.g. `netCDF::NcFile::read` or `netCDF::NcFile::replace` for reading and writing, respectively.
- netCDF *group*: a netCDF file is organised into several *groups*, which can be nested inside each other to an arbitrary degree. In the C++ interface used by SMS++, each group corresponds to an object of type `netCDF::NcGroup`; note that `netCDF::NcFile` derives from `netCDF::NcGroup`, and therefore all capabilities of the group are shared by the file (but not vice-versa). Each *group* has a unique string name by which it can retrieved; `netCDF::NcGroup` (and, therefore, `netCDF::NcFile`) has a method `getGroup()` taking as input a `std::string` with the group name and returning an `netCDF::NcGroup` object containing the *child group* with the given name (if any) of that group.
- netCDF *attribute*: is a simple data point specific of a netCDF group, having a unique string name by which it can retrieved and a simple value (`int`, `double`, `std::string`, ...). In the C++ interface used by SMS++, it is represented by a `netCDF::NcGroupAtt` object, which can be obtained by its enclosing `netCDF::NcGroup` via the method `getAtt()` taking in input its name. Then, the method(s) `getValue()` (available with different signatures corresponding to the different supported basic data types) allows to retrieve its value.

- NETCDF *dimension*: it basically represent the different indices over which the actual data is indexed, each of which is given a name. In the C++ interface used by SMS++, it is represented by a `netCDF::NcDim` object, which can be obtained by its enclosing `netCDF::NcGroup` via the method `getDim()` taking in input its name. The *dimension* (a `size_t` value) is returned by the method `getSize()`.
- NETCDF *variable*: this is the container for the "complex" data of a *groups*. It has a name (`std::string`), a basic data type (`int`, `double`, `std::string`, ...) and in principle any number of *dimensions*. In the C++ interface used by SMS++, it is represented by a `netCDF::NcVar` object, which can be obtained by its enclosing `netCDF::NcGroup` via the method `getVar()` taking in input its name. Once the `netCDF::NcVar` object is obtained, individual values of the *variable* can be read by its method(s) `getVar()` (available with different signatures corresponding to the different supported basic data types), taking a `std::vector<size_t>` as long as the number of *dimensions* of the *variable* (obtainable e.g. with its method `NgetDimCount()`).

The above components allow to read and write structured data files with basically any nested format, which makes NETCDF ideal for representing the data characterizing SMS++, as discussed below.

3 General SMS++ file structure

In SMS++, files can describe two different kinds of objects:

1. `Block` objects, representing the data characterizing a specific instance of a (structured) optimization problem to be solved;
2. `Configuration` objects, representing parameters of the solution process, either pertaining to a `Block` (say, which of the different available formulations of the optimization problem should be used), or to a `Solver` used to actually perform the optimization process.

Both objects have `serialize()` and `deserialize()` methods allowing to save the current state to a given NETCDF *file*/*group*, as well as *factories* allowing to create a new object from scratch out of a given NETCDF *file*/*group*.

The issue of SMS++ is that an optimization problem is represented by a `Block`, which can have any number of sub-`Block` (recursively) representing parts of the problem with a specific structure. Yet, "specific structure" here may mean that the enclosing `Block` can make some assumptions on what kind of sub-`Block` it contains, but it may not know exactly their type. In C++ parlance, `Block` is an abstract base class, and specific optimization problems are represented by specific derived classes (`:Block`). A given `:Block` can make assumption on the type of its sub-`Block`, but there can be many different variants of it, represented by different derived classes from the same base one. Thus, clearly the data-reading must be organized in such a way that each `:Block` is responsible for retrieving that which is necessary to it without making assumptions about the global organization of the file. This does not only apply to the data of the instance, but also to *configuration* options that a `:Block` may have. In fact, several aspect of the `:Block` representing a given problem (which formulation is used, if and how variables and constraint of the formulation are dynamically generated, which parts of the primal/dual solution are saved, ...) can be implemented in different ways. Note that these decisions are in principle independent from the specific instance, and therefore are conceptually to be stored in, and retrieved from, a separate container than that holding the data of the `Block`; this is why the `Configuration` class is provided. Since `Block` is a tree-shaped nested data structure, `Configuration` also have to be such. Clearly, NETCDF perfectly suit this kind of requirement; in particular, the standing assumption for both classes is that

the data of one object (`Block`/`Configuration`) *is all to be found in the same* NETCDF *group;*
the data of its "sub-objects" (sub-`Block`/sub-`Configuration`) is to be found in child groups of the group.

Hence, the format of a SMS++ NETCDF file depends on whether it holds `Block` information, `Configuration` information, or both. For the latter, there are actually two separate kind of `Configuration` information that can be attached to a `:Block`. The first describes the above-mentioned options that are inherent to the `:Block` itself, which are described by the specific derived class `BlockConfig` of `Configuration`. However, a `:Block` need be solved, which is done by `Solver` objects; each `Block` can have an arbitrary number of `Solver` "registered" with it. A `Solver` can be an arbitrarily complex solution process; in particular, the structure of `Block` immediately implies that one can register `Solver` with any sub-`Block`, which is actually useful e.g. to decomposition algorithms. Hence, not only each `:Solver` can have an arbitrarily large set of algorithmic configuration options; an arbitrarily large set

of Solver can be attached to a Block and to all of its sub-Block, recursively. Thus, the different derived class BlockSolverConfig of Configuration is provided to describe all this information. Note, again, that this is conceptually separate from both the specific instance encoded in the :Block and the choice of the Block-specific options encoded in the BlockConfig, which justifies why a separate container is used.

To allow holding all this different information, a SMS++ NETCDF file has the following general structure. It must contain (in the netCDF::NcFile object, which is also a netCDF::NcGroup) an *attribute* SMS++_file_type of int type which specifies which of three different kinds of file it is. The values are encoded in the enum sm spp_netCDF_file_type defined in SMSTypedefs.h, as follows:

- eProbFile = 0: the *file* (which is also a *group*) has any number of child *groups* with names Prob_0, Prob_1, ... In turn, each child *group* has *exactly three* child *groups* with names Block, BlockConfig and Block Solver, respectively. The first is intended to contain the serialization of a :Block, the second the serialization of a :BlockConfig of the same :Block, and the third the serialization of a :BlockSolverConfig of the same :Block, although any of the three can in principle be empty. If any of the child is not empty, it *must* necessarily contain all the information necessary to reconstruct the specific instance.
- eBlockFile = 1: the *file* (which is also a *group*) has any number of child *groups* with names Block_0, Block_1, ... Each child *group* contains the serialization of a :Block.
- eConfigFile = 2: the *file* (which is also a *group*) has any number of child *groups* with names Config_0, Config_1, ... Each child *group* contains the serialization of a :Configuration. Note that no distinction is required between a :BlockConfig and a :BlockSolverConfig, since they are both derived from Config uration.

Note that (some variants of) the deserialize() methods of Block/ Configuration allow to directly specify which of the different objects stored in the file to read via its index i in the "name".

To completely describe the file formats we now have to describe the format of the NETCDF *group* as far as the base Block and Configuration (for the latter, actually both sub-classes BlockConfig and BlockSolverConfig) are concerned, i.e., the (possibly, small) part of the contents that do not depend on the specific derived class.

3.1 Block description

The only content that any NETCDF *group* describing a :Block must necessarily have is a string *attribute* with name type containing the classname of the object. This must be *exactly* what is returned by the protected virtual method classname(), since it is used in the *factory* when deserializing the object. Note that template classes derived from Block are allowed, with the standard notation class_name<template_parameters>. However, due to a technical issue about using macros (see the comments to SMSpp_insert_in_factory_cpp_* in SMSTypedefs.h), *the name given to the class must not contain any "," (comma).* That is, something like "myBlock< std::pair< int , int > >" *cannot* be used when defining the Block. The typical way is to resort to a using declaration, i.e.,

```
using myBlock_i_i = myBlock< std::pair< int , int > >;
SMSpp_insert_in_factory_cpp_1_t( myBlock_i_i );
```

This implies that the type of that Block as returned by classname(), and therefore the one that has to be used in the NETCDF file, must necessarily be "myBlock_i_i". The exact string will typically be defined in the .cpp file where the Block is implemented, precisely in the corresponding invocation of the macro SMSpp_insert_in_factory_cpp_*. In all non-complicated-template cases, the name should just be the only possible form in which the classname can be written.

Besides the mandatory type *attribute*, all Block support an optional name *attribute* of string type. If present, this is supposed to contain the "name" of the Block, as returned by the name() method; if not present (and not set in-memory via set_name()), name() will return an empty string. This is not a classname, but rather a string describing something about the Block that is supposedly useful when printing/debugging it.

3.2 Configuration description

The only content that any NETCDF *group* describing a :Configuration must necessarily have is a string *attribute* with name type containing the classname of the object. This must be exactly what is returned by the protected

virtual method `classname()`, since it is used in the *factory* when deserializing the object. Note that template classes derived from `Configuration` are allowed, with the standard notation `class_name<template_parameters>`; see for instance the template `SimpleConfiguration<>` class. However, the same issue with "," (commas) not being allowed to the class name as in `Block` applies here, see §3.1 for more details.

3.2.1 `BlockConfig` description

Besides the mandatory `type` attribute of any `:Configuration`, a SMS++ NETCDF *group* for a `BlockConfig` can contain the following:

- the *attribute* `diff` of `netCDF::NcInt` type telling if the information in it has to be taken as "the configuration to be set" ($\texttt{diff} \leq 0$) or as "the changes to be made from the current configuration" ($\texttt{diff} > 0$); this attribute is optional: if it is not provided, then $\texttt{diff} = 1$ is assumed;
- the *group* `static_constraints` containing a `Configuration` object for the static constraints of the `Block`;
- the *group* `dynamic_constraints` containing a `Configuration` object for the dynamic constraints of the `Block`;
- the *group* `static_variables` containing a `Configuration` object for the static variables of the `Block`;
- the *group* `dynamic_variables` containing a `Configuration` object for the dynamic variables of the `Block`;
- the *group* `objective` containing a `Configuration` object for the objective of the `Block`;
- the *group* `is_feasible` containing a `Configuration` object for the `is_feasible()` method of the `Block`;
- the *group* `is_optimal` containing a `Configuration` object for the `is_optimal()` method of the `Block`;
- the *group* `solution` containing a `Configuration` object for the methods of the `Block` dealing with solutions (`get_Solution()` and `map_[back/forward]_solution()`);
- the *group* `extra` containing a `Configuration` object, which has no direct use in the base `Block` class, but is added so that derived classes can put there any configuration information without having to define further derived classes form `BlockConfig` (which, however, they can still do if they want).

All these *groups* are optional. If a *group* is not provided, the corresponding field of the class is filled with a `nullptr`, indicating that the "default" configuration (whatever that may mean for the `:Block` in question) has to be used.

3.2.1.1 `RBlockConfig` description

Besides the mandatory `type` attribute of any `:Configuration`, a SMS++ NETCDF *group* for a `RBlockConfig` should also contain the following:

- all that is needed to describe a `BlockConfig` (see 3.2.1):
- the *dimension* `n_sub_Block` containing the number of `BlockConfig` descriptions for the sub-`Block` of the current `Block`; this *dimension* is optional; if it is not provided, then $\texttt{n\_sub\_Block} = 0$ is assumed.
- With n being the size of `n_sub_Block`, n *groups*, with name `sub-BlockConfig_`i for all $i = 0, \ldots, n-1$, containing each the description of a `BlockConfig` for one of the sub-`Block` of the current `:Block`. Each of these *groups* is optional. If a *group* is absent then the pointer to the `BlockConfig` for the corresponding sub-`Block` is considered to be a `nullptr` (default configuration).
- With n being the size of `n_sub_Block`, a one-dimensional *variable* with name `sub-Block-id`, of size n and type `netCDF::NcString`, containing the identification of the sub-`Block` such that `sub-BlockConfig_`i contains the `BlockConfig` for the sub-`Block` whose identification is `sub-Block-id[ `i` ]` for all $i = 0, \ldots, n-1$. The identification of the sub-`Block` can be either its name (see `Block::name()`) or its index in the list of sub-`Block` of its father `Block`. This *variable* is optional. If it is not provided, then the i-th `BlockConfig`

is associated with the i-th sub-`Block` of the `Block` for all $i = 0, \ldots, n-1$ (i.e., i is taken as the index of the sub-`Block` and `sub-Block-id[ i ]` is assumed to be i).

> **Note:** If the name of the `Block` is used as its identification, then the first character of this name cannot be a digit.

3.2.1.2 `CBlockConfig` description

Besides the mandatory `type` attribute of any `:Configuration`, a SMS++ NETCDF *group* for a `CBlockConfig` should also contain the following:

- all that is needed to describe a `BlockConfig` (see 3.2.1):
- the *dimension* `n_Config_Constraint` containing the number of `ComputeConfig` descriptions associated with the `Constraint` of the current `Block`; this *dimension* is optional; if it is not provided, then `n_Config_Constraint` $= 0$ is assumed.
- with p being the size of `n_Config_Constraint`, a one-dimensional *variable* called `Constraint_group_id`, of size p and type `netCDF::NcString`, containing the identification of the group of `Constraints` that require a `ComputeConfig`. The i-th element of this vector, `Constraint_group_id[ i ]`, is the identification of the group to which the i-th `Constraint` belongs. For each $i = 0, \ldots, p-1$, `Constraint_group_id[ i ]` can be indicated in two ways: it is either (i) the name of the group of `Constraint` (see `Block::get_s_const_name()` and `Block::get_d_const_name()`) or the index of the group as defined in `Block::ConstraintID`.

  > **Note:** If a `Constraint` group is being identified using the name of the group (rather than the index of the group), then
  >
  > – the first character of this name cannot be a digit;
  >
  > – the static group has priority over the dynamic group of `Constraint`: if there is a group of static `Constraint` with the given name, then this group is considered. Otherwise, the group of dynamic `Constraint` with that name is considered.

  This *variable* is mandatory if `n_Config_Constraint` > 0.
- with p being the size of `n_Config_Constraint`, a one-dimensional *variable* called `Constraint_index`, of size p and type `netCDF::NcUint`, containing the index of the `Constraint` that require a `ComputeConfig`. The i-th element of this vector, `Constraint_index[ i ]`, is the index of the i-th `Constraint` (which belongs to the group indicated by `Constraint_group_id[ i ]`). See `Block::ConstraintID` for the definition of the index of a `Constraint` in a group. This *variable* is optional. If it is not provided, then `Constraint_index[ i ]` $= i$ for all $i = 0, \ldots, p-1$ is assumed.
- p *groups*, with name `Config_Constraint_`i for all $i = 0, \ldots, p-1$, containing each the description of a `ComputeConfig` associated with the i-th `Constraint` indicated by the pair (`Constraint_group_id[ i ]`, `Constraint_index[ i ]`); these *groups* are optional; if `Config_Constraint_`i is not provided, then `nullptr` (default configuration) is assumed for the i-th `Constraint`.

3.2.1.3 `OBlockConfig` description

Besides the mandatory `type` attribute of any `:Configuration`, a SMS++ NETCDF *group* for a `OBlockConfig` should also contain the following:

- all that is needed to describe a `BlockConfig` (see 3.2.1):
- a *group* with name `Config_Objective`, containing the description of a `ComputeConfig` associated with the `Objective` of the current `Block`; this *group* is optional; if it is not provided, then `nullptr` (default configuration) is assumed.

3.2.1.4 `OCBlockConfig` description

Besides the mandatory `type` attribute of any `:Configuration`, a SMS++ NETCDF *group* for a `OCBlockConfig` should also contain the following:

- all that is needed to describe a `BlockConfig` (see 3.2.1):
- all that is needed to describe a `OBlockConfig` (see 3.2.1.3):
- all that is needed to describe a `CBlockConfig` (see 3.2.1.2).

3.2.1.5 `ORBlockConfig` description

Besides the mandatory `type` attribute of any `:Configuration`, a SMS++ NETCDF *group* for a `ORBlockConfig` should also contain the following:

- all that is needed to describe a `BlockConfig` (see 3.2.1):
- all that is needed to describe a `OBlockConfig` (see 3.2.1.3):
- all that is needed to describe a `RBlockConfig` (see 3.2.1.1).

3.2.1.6 `CRBlockConfig` description

Besides the mandatory `type` attribute of any `:Configuration`, a SMS++ NETCDF *group* for a `CRBlockConfig` should also contain the following:

- all that is needed to describe a `BlockConfig` (see 3.2.1):
- all that is needed to describe a `CBlockConfig` (see 3.2.1.2):
- all that is needed to describe a `RBlockConfig` (see 3.2.1.1).

3.2.1.7 `OCRBlockConfig` description

Besides the mandatory `type` attribute of any `:Configuration`, a SMS++ NETCDF *group* for a `OCRBlockConfig` should also contain the following:

- all that is needed to describe a `BlockConfig` (see 3.2.1);
- all that is needed to describe a `OBlockConfig` (see 3.2.1.3);
- all that is needed to describe a `CBlockConfig` (see 3.2.1.2):
- all that is needed to describe a `RBlockConfig` (see 3.2.1.1).

3.2.2 `BlockSolverConfig` description

Besides the mandatory `type` attribute of any `:Configuration`, a SMS++ NETCDF *group* for a `BlockSolverConfig` can contain the following:

- the *attribute* `diff` of `netCDF::NcInt` type telling if the information in it has to be taken as "the configuration to be set" ($\texttt{diff} \leq 0$) or as "the changes to be made from the current configuration" ($\texttt{diff} > 0$); this attribute is optional, if it is not provided, then $\texttt{diff} = 0$ is assumed;
- the *dimension* `n_SolverConfig` containing the number of `Solver` that are to be attached to the `Block`, and therefore the number of their `SolverConfig` objects; this dimension is optional, if it is not provided, then $\texttt{n\_SolverConfig} = 0$ is considered;

- the *variable* SolverNames, of type netCDF::NcString and indexed over the *dimension* n_SolverConfig; the i-th entry of the variable is assumed to contain the classname of a :Solver object to be attached to the Block (this must be exact, i.e., exactly as returned by the protected virtual method Solver::classname(), since it is used in the *factory* when creating the object; this variable is mandatory if n_SolverConfig > 0;
- with n being the size of n_SolverConfig, n *groups*, with name SolverConfig_i for all $i = 0, \ldots, n-1$, containing each the description of a ComputeConfig object for the i-th :Solver; these groups are optional; if SolverConfig_i is not provided, then nullptr is considered for the i-th ComputeConfig.

Note that the :Configuration objects to be found in the *groups* SolverConfig_i are assumed to be of the specific type ComputeConfig, whose format is described in 3.2.3.

3.2.2.1 RBlockSolverConfig description

Besides the mandatory type attribute of any :Configuration, and the *dimensions*, *variables*, and *groups* of a BlockSolverConfig, a SMS++ NETCDF *group* for a RBlockSolverConfig can contain the following:

- the *dimension* n_BlockSolverConfig containing the number of BlockSolverConfig descriptions for the sub-Block of the current Block; it is optional; if it is not provided, then we assume n_BlockSolverConfig $= 0$.
- with m being the size of n_BlockSolverConfig, m *groups*, with name BlockSolverConfig_i for all $i = 0, \ldots, m-1$, containing each the description of a BlockSolverConfig for one of the sub-Block of the current Block. If some group BlockSolverConfig_i does not exist, nullptr is used;
- With n being the size of n_BlockSolverConfig, a one-dimensional *variable* with name sub-Block-id, of size n and type netCDF::NcString, containing the identification of the sub-Block such that BlockSolver Config_i contains the BlockSolverConfig for the sub-Block whose identification is sub-Block-id[i] for all $i = 0, \ldots, n-1$. The identification of the sub-Block can be either its name (see Block::name()) or its index in the list of sub-Block of its father Block. This *variable* is optional. If it is not provided, then the i-th BlockSolverConfig is associated with the i-th sub-Block of the Block for all $i = 0, \ldots, n-1$ (i.e., i is taken as the index of the sub-Block and sub-Block-id[i] is assumed to be i).

  > **Note:** If the name of the Block is used as its identification, then the first character of this name cannot be a digit.

  The individual *groups* BlockSolverConfig_i are optional. If BlockSolverConfig_i is not provided, then nullptr is considered. Note that the size of the vector of sub-BlockSolverConfig is allowed to be different than the number of sub-Block.

3.2.3 ComputeConfig description

ComputeConfig (defined in ThinComputeInterface.h) is a class of :Configuration objects specifically tailored for representing sets of algorithmic parameters of solution algorithms, like the ones presumably required by a :Solver. Besides the mandatory type attribute of any :Configuration, a SMS++ NETCDF *group* for a ComputeConfig can contain the following:

- the *attribute* diff of int type containing the value for the f_diff field of the ComputeConfig (basically, a bool telling if the information in it has to be taken as "the configuration to be set" or as "the changes to be made from the current configuration");
- the *dimension* num_int_par containing the number of int parameters;
- the *variable* int_par_names, of type string and indexed over the *dimension* num_int_par; the i-th entry of the variable is assumed to contain the string name of an int parameter;
- the *variable* int_par_vals, of type int and indexed over the *dimension* num_int_par; the i-th entry of the variable is assumed to contain the value of the int parameter whose string name is to be found in the i-th entry of int_par_names;

- the *dimension* num_dbl_par containing the number of double parameters;
- the *variable* dbl_par_names, of type string and indexed over the *dimension* num_dbl_par; the i-th entry of the variable is assumed to contain the string name of a double parameter;
- the *variable* dbl_par_vals, of type double and indexed over the *dimension* num_dbl_par; the i-th entry of the variable is assumed to contain the value of the double parameter whose string name is to be found in the i-th entry of dbl_par_names;
- the *dimension* num_str_par containing the number of string parameters;
- the *variable* str_par_names, of type string and indexed over the *dimension* num_str_par; the i-th entry of the variable is assumed to contain the string name of a string parameter;
- the *variable* str_par_vals, of type string and indexed over the *dimension* num_str_par; the i-th entry of the variable is assumed to contain the value of the string parameter whose string name is to be found in the i-th entry of str_par_names;
- the *group* extra containing a Configuration object, which has no direct use in the base ComputeConfig class, but is added so that derived classes can put there any configuration information without having to define further derived classes form ComputeConfig (which, however, they can still do if they want).

The three *dimensions* num_*_par are mandatory. If any of these is zero, the corresponding variables *_par_names and *_par_vals may not be defined. The extra *group* may not exist, in which case the corresponding Configuration object is set to a nullptr.

3.3 Auxiliary structures formats

This section presents the description of some auxiliary structures that are necessary to describe some :Block. The AbstractPath, presented in §3.3.1, is used to specify a path from a Block to some SMS++ element (Block, Constraint, Variable, Objective, or Function) defined in that Block or in any of their nested Block, recursively. The SimpleDataMapping, presented in §3.3.2, offers a mechanism for changing the data of a Block.

3.3.1 AbstractPath

The AbstractPath represents a path from a Block, here referred to as the reference Block, to one of its elements (the target element): a Block, a Constraint, a Variable, an Objective, or a Function. The target element can directly belong to the reference Block itself (even be the reference Block itself) or belong to any of its sons, recursively. The reference Block is not explicitly represented in the path. In fact, the representation of the path is independent from the reference Block. For the path to be meaningful, the reference Block should be clear from the context. The reference Block must be available when the path is constructed and when the target element is retrieved. Furthermore, the type of the target element cannot always be inferred from the path. The type of the target element, therefore, must also be known from the context.

The AbstractPath is particularly useful for the serialization and deserialization of pointers to objects. If an object stores pointers to other objects (for example, a Function has a set of pointers to Variable, others have pointers to Block), these pointers cannot be serialized and deserialized as such. Rather, an "abstract representation" of these pointers has to be serialized, from which the pointers can be reconstructed at deserialization time; this is the facility that AbstractPath offers. The fact that the representation of the path is independent from the reference Block facilitates its serialization and deserialization, and makes it possible to use the same path to target different objects (say, two Constraint "in the same position" in two "twin" Block can be represented by the same AbstractPath).

The path is defined as a sequence of nodes. Each node has one of the following types:

- 'O', if the node is associated with an Objective;
- 'B', if the node is associated with a Block;
- 'C', if the node is associated with a static Constraint;
- 'c', if the node is associated with a dynamic Constraint;

- 'V', if the node is associated with a static `Variable`;
- 'v', if the node is associated with a dynamic `Variable`.

Notice that, for `Constraint` and `Variable`, an upper case letter indicates that the element is static, while a lower case letter indicates that the element is dynamic. Also notice that there is no node associated with a `Function`, but this does not prevent one from constructing a path to a `Function`. Although the nodes in the path are stored in forward order, i.e., the first node is the origin of the path and the last one is the destination, it is easier to understand the path if we look at it backwards.

Consider the path from some reference `Block` to some `Variable`. The last node in this path necessarily has the 'V' or 'v' type, indicating this is a path to a `Variable`. This `Variable` belongs to some `Block` and it is either static or dynamic. This last node has all information needed to retrieve this `Variable` from its father `Block`: an indication of whether the `Variable` is static or dynamic, the index of the group to which it belongs, and the index of the `Variable` within that group.

> **Note:** The index of a `Variable` (or `Constraint`) within a group is a single number and may not be entirely obvious which number it should be (specially for multidimensional arrays and (multidimensional) arrays of lists). Please refer to the §3.3.1.1 for an explanation of the index of a `Variable` (or `Constraint`) within a group.

A 'V' or 'v' node is always preceded by a 'B' node, unless it is the only node in the path. If the path has only a single node, which is a 'V' or 'v' node, then the `Variable` this path refers to is defined in the reference `Block`. In other words, if the target `Variable` of the path belongs to the reference `Block`, then the path is formed by a single node whose type is either 'V' or 'v'.

A 'B' node is associated with a `Block` and may contain the index of this `Block` in the vector of nested `Blocks` of its father `Block`. There is one case in which this node does not have this index, which is when the node is associated with the reference `Block`. Since the reference `Block` is the root of the three that contains the path, no allusion to the father of the reference `Block` must be made. In this case, the index has the value $+\infty$. If the index of a 'B' node is not $+\infty$, then this node is necessarily preceded by another 'B' node, which is associated with the father of that `Block`.

An 'O' node, which is associated with an `Objective`, is either preceded by a 'B' node (which is associated with the `Block` that owns that `Objective`) or is the only node in the path. If it is the last node in the path, then the target element is either this `Objective` or the `Function` in that `Objective` (in which case that `Objective` is an `FRealObjective`). The type of the target element must be known from the context. If this is not the last node in the path, then the type of the next node in the path is 'B' and it is associated with the inner `Block` of either a `BendersBFunction` or a `LagBFunction` which is the `Function` of that `Objective` (and thus that `Objective` is actually an `FRealObjective`).

Finally, a 'C' or 'c' node, which is associated with a `Constraint`, has characteristics pertaining both the 'V' (and 'v') and the 'O' nodes. Like the 'V' (and 'v') node, it has an indication of whether it is static or not, the index of the group to which it belongs, and the index of the `Constraint` within that group (exactly as defined for `Variable`). Like an 'O' node, a 'C' or 'c' node is either preceded by a 'B' node (which is associated with the `Block` that owns that `Constraint`) or is the only node in the path. If it is the last node in the path, then the target element is either this `Constraint` or the `Function` in that `Constraint` (in which case that `Constraint` is an `FRowConstraint`). The type of the target element must be known from the context. If this is not the last node in the path, then the type of the next node in the path is 'B' and it is associated with the inner `Block` of either a `BendersBFunction` or a `LagBFunction` which is the `Function` of that `Constraint` (and thus that `Constraint` is actually an `FRowConstraint`).

Besides the description of an `AbstractPath`, we also provide a description of a vector of `AbstractPath`. In §3.3.1.1 we present the description of a single `AbstractPath` and in §3.3.1.2 we present the description of a vector of `AbstractPath`.

3.3.1.1 A single `AbstractPath`

An `AbstractPath` is described by the following NETCDF *variables* and *dimensions*. These do not need to be in a specific NETCDF *group*, unless of course one has to specify more than one `AbstractPath` in a *group*, in which

case it is necessary to make specific *groups* because the name of a *variable*/*dimension* cannot be duplicated in a given *group*.

- the *dimension* `TotalLength` containing the number N of nodes in the path;
- the *variable* `PathNodeTypes`, of type `netCDF::NcChar` and indexed over the *dimension* `TotalLength`, contains the types of the nodes in the path;
- the *variable* `PathGroupIndices`, of type `netCDF::NcUint` and indexed over the *dimension* `TotalLength`, contains the group indices associated with each node in the path;
- the *variable* `PathElementIndices`, of type `netCDF::NcUint` and indexed over the *dimension* `TotalLength`, contains the element indices associated with each node in the path. This *variable* is optional. If it is not provided, then `PathElementIndices[i]` is assumed to be `inf` for all $i \in \{0, \ldots, \texttt{TotalLength} - 1\}$.

Note: All indices mentioned here belong to zero-based numbered sequences, that is, sequences whose first element is 0.

The i-th element in each of these arrays is associated with the i-th node in the path, for $i \in \{0, \ldots, N-1\}$. The type of a node may be indicated by any of the following letters:

- 'O', if the node is associated with an `Objective`;
- 'B', if the node is associated with a `Block`;
- 'C', if the node is associated with a static `Constraint`;
- 'c', if the node is associated with a dynamic `Constraint`;
- 'V', if the node is associated with a static `Variable`;
- 'v', if the node is associated with a dynamic `Variable`.

Notice that, for `Constraint` and `Variable`, an upper case letter indicates that the element is static, while a lower case letter indicates that the element is dynamic.

For a node whose type is 'O', i.e., representing a node associated with an `Objective`, the values stored in the variables `PathGroupIndices` and `PathElementIndices` are meaningless. If it is the last node in the path, then the path refers to an `Objective`. Otherwise, the next node in the path must be of type 'B'.

For node whose type is 'B', i.e., representing a node associated with a `Block` B, the values stored in the variable `PathElementIndices` are meaningless. If this is the i-th node in the path with $i < N-1$, i.e., it is not the last node of the path, the value stored in `PathGroupIndices[i]` is the index of the sub-`Block` of `Block` B which is the next node in the path. If $i = N-1$, i.e., it is the last node in the path, then the destination of the path is a `Block` which must be

- the `Block` B itself if `PathGroupIndices[i]` $= +\infty$;
- the j-th sub-`Block` of `Block` B, where $j =$ `PathGroupIndices[i]`.

If the i-th node has type 'C' or 'c', representing a `Constraint`, or 'V' or 'v', representing a `Variable`, the variable `PathGroupIndices[i]` stores the index of the (static or dynamic) group to which the element belongs. The variable `PathElementIndices[i]` stores the index of the element in that group.

A static group can be one of three types:

1. It is a single `Constraint` or `Variable`;
2. It is a vector of `Constraint` or `Variable`;
3. It is a multidimensional array of `Constraint` or `Variable`.

In the first case, in which the group is a single `Constraint` or `Variable`, the value of `PathElementIndices[i]` is 0. In the second case, in which the group is a vector of `Constraint` or `Variable`, `PathElementIndices[i]` is the index of the element in that vector. In the last case, in which the group is a multidimensional array of `Constraint` or `Variable`, `PathElementIndices[i]` is the index of the element in the vectorized multidimensional array in row-major layout. For instance, if the multidimensional array has two dimensions with sizes m and n, respectively, then the element at position (p, q) would have an element index equal to $np + q$ (recall the indices start from 0). In general, for a multidimensional array with k dimensions with sizes $(n_0, \ldots, n_{k-1})$, the element at position $(i_0, \ldots, i_{k-1})$ would have an element index equal to

$$\sum_{r=0}^{k-1} \left(\prod_{s=r+1}^{k-1} n_s \right) i_r. \tag{1}$$

A dynamic group can be one of three types:

1. It is list of `Constraint` or `Variable`;
2. It is a vector of lists of `Constraint` or `Variable`;
3. It is a multidimensional array of lists of `Constraint` or `Variable`.

In the first case, in which the group is a list of `Constraint` or `Variable`, `PathElementIndices[i]` is the index of the element in that list. In the second case, in which the group is a vector of lists of `Constraint` or `Variable`, for an element at position j of the k-th list of the vector, `PathElementIndices[i]` is given by

$$j + \sum_{t=0}^{k-1} s_t$$

where s_t is the number of elements in the t-th list of the vector. The last case is analogous.

3.3.1.2 Vector of `AbstractPath`

A vector of `AbstractPath` is described by the following NETCDF *variables* and *dimensions*. These do not need to be in a specific NETCDF *group*, unless of course one has to specify more than one vector of `AbstractPath` in a *group*, in which case it is necessary to make specific *groups* because the name of a *variable*/*dimension* cannot be duplicated in a given *group*.

- the *dimension* `PathDim`;
- the *dimension* `TotalLength`;
- the *variable* `PathStart`, of type `netCDF::NcUint` and indexed over `PathDim`;
- the *variable* `PathNodeTypes`, of type `netCDF::NcChar` and indexed over the *dimension* `TotalLength`;
- the *variable* `PathGroupIndices`, of type `netCDF::NcUint` and indexed over the *dimension* `TotalLength`;
- the *variable* `PathElementIndices`, of type `netCDF::NcUint` and indexed over the *dimension* `TotalLength`. This *variable* is optional. If it is not provided, then `PathElementIndices[i]` is assumed to be `inf` for all $i \in \{0, \ldots, \texttt{TotalLength} - 1\}$.

The format of a vector of `AbstractPath` is similar to that of a single `AbstractPath` with a few differences. Firstly, there is a *dimension* called `PathDim` which contains the number of paths that are described in that group. Secondly, there is a one-dimensional *variable* called `PathStart`, indexed over `PathDim`, which contains the index where the description of each `AbstractPath` starts. Then there are the one-dimensional *variables* `PathNodeTypes`, `PathGroupIndices`, and `PathElementIndices`, which store the types of nodes in the paths, the group indices, and the element indices, respectively. The description of the k-th `AbstractPath`, for $k <$ `PathDim` $- 1$ is given in those arrays between the indices `PathStart[k]` and `PathStart[k+1]`, i.e., in

```
(PathNodeTypes[ PathStart[ k ] ],...,PathNodeTypes[ PathStart[ k+1 ] - 1 ]),
```

(PathGroupIndices[PathStart[k]],...,PathGroupIndices[PathStart[k+1] - 1])

and

(PathElementIndices[PathStart[k]],...,PathElementIndices[PathStart[k+1] - 1]).

The description of the last path is given between the indices PathStart[PathDim - 1] and TotalLength - 1, where TotalLength is a *dimension* containing the size of the arrays PathNodeTypes, PathGroupIndices, and PathElementIndices (and, therefore, these arrays are indexed over TotalLength).

The paths are represented in the same format as specified for a single AbstractPath, except that here they are concatenated in the arrays PathNodeTypes, PathGroupIndices, and PathElementIndices.

3.3.2 SimpleDataMapping

DataMapping defines an interface for all types of data mappings. The idea of a data mapping is to allow, in particular, to map the values given by a vector of double into the data of some object. Among its functions, it has the set_data() pair of functions, which have the following signature:

```
virtual void set_data( const std::vector< double > & data ,
                       c_ModParam issueMod = eModBlck ,
                       c_ModParam issueAMod = eModBlck ) const;

virtual void set_data( const Eigen::ArrayXd & data ,
                       c_ModParam issueMod = eModBlck ,
                       c_ModParam issueAMod = eModBlck ) const;
```

The idea of these functions is that the values of some data of an object can be modified considering the given data parameter. Typically, a DataMapping could be used to set the data of a Block. In this case, a pointer to that Block must be available. Pointers to a Block can be serialized and deserialized considering its AbstractPath, which is relative to some reference Block.

SimpleDataMapping is a template class that derives from DataMapping and is used to define some common kinds of data mapping. We define two vectors: the "large" one and the "small" one. The "large" vector refers to the vector that is given as input to the set_data() method. This is the vector containing all the data that can be used by the SimpleDataMapping. The "small" vector is a vector formed by a subset of the elements of the "large" vector. This is the vector that will effectively be used to perform some computation. This computation is typically the task of changing the data of some object based on this "small" vector. There is a mapping defined by the SetFrom set that specifies which elements of the "large" vector are used to compose the "small" vector. This SetFrom set contains the indices of these elements in the "large" vector. The "small" vector is the one that will typically impact the data of some object. The SetTo set can be used to specify which part of this data is affected. Actually, SetFrom and SetTo are ordered multisets, but we will refer to them as sets for simplicity.

As an example, consider the case in which the "large" vector contains data related to costs and capacities of arcs of a network. Suppose this network is represented by a fictitious class Network. A SimpleDataMapping could be used to set the capacities of the arcs of a Network object considering the data provided by this "large" vector. Some elements of this "large" vector would be extracted and form a "small" vector containing the capacities of some arcs. The indices of the elements that are extracted from the "large" vector are specified by the SetFrom set. This set could be, for instance, the set $\{0, 3, 8, 11\}$. This means that the elements at positions $0, 3, 8$, and 11 in the "large" vector are selected to form a "small" vector with four elements. This "small" vector would then be used to change the capacities of some arcs of the Network object. The arcs whose capacities would be modified could be specified by the SetTo set. This could be the set $[2, 6)$, for instance, stating that the arcs with indices $2, 3, 4$, and 5 would have their capacities changed according to the "small" vector.

Usually, the SetFrom and the SetTo sets will have the same cardinality, so that the i-th element of the SetFrom set will be associated with the i-th element of the SetTo set. However, the SetFrom set is also allowed to be smaller than the SetTo set. In this case, the cardinality of the SetTo set must be a positive multiple of the cardinality of the SetFrom set and the i-th element of the SetTo set will be associated with the element of the SetFrom set located at position $\lfloor i/r \rfloor$, where r is the ratio between the cardinalities of the SetTo and SetFrom sets.

Besides the SetFrom and SetTo sets, the SimpleDataMapping also has a pointer to a function, which is invoked within the set_data() method. This is a function that receives, in particular, a pointer to a Block, the "small" vector, and the SetTo set. If the SetTo set is a Block::Subset, then the type of this function is

```
Block::FunctionType< typename std::vector< DataType >::const_iterator ,
                     SetTo && , bool >
```

If the SetTo set is a Block::Range, then the type of this function is

```
Block::FunctionType< typename std::vector< DataType >::const_iterator ,
                     const SetTo & >
```

A Range represents the closed-open interval $[a, b)$ set of indices (integers), determined by the integers a and b. A Subset represents a set of indices by explicitly listing its elements. Please refer to the definition of Block::FunctionType for completely understanding the type of this function.

In the network example above, this function could be, for instance,

```
void set_capacities( Network * network ,
                     std::vector<double>::const_iterator capacities ,
                     const Range & indices ,
                     c_ModParam , c_ModParam );
```

Ignoring the details of the type of this function, this is a function that receives a pointer to a Network object, a vector of capacities, and a Range of indices. This function could be responsible for changing the capacities of the arcs (of the given Network object) specified by the indices parameter according to the given capacities.

As you can see, the function associated with a SimpleDataMapping receives a pointer to a Block as its first parameter. This is a pointer to the caller object; the object that "invokes" the function. In the network example, this would be a pointer to a Network object whose capacities would be modified.

Finally, the SimpleDataMapping is also determined by the type of the data of the "small" vector, the DataType.

Notice that a SimpleDataMapping is general enough in that it is not only meant to change the data of some object, but perform arbitrary computation defined by the function associated with this SimpleDataMapping.

In summary, a SimpleDataMapping has the following template parameters:

- SetFrom: This is the type of the set that selects the appropriate data from data vector. It must be either Block::Range or Block::Subset.
- SetTo: This is the type of the set that indicates which part of the data of the caller object that is affected. It must be either Block::Range or Block::Set.
- DataType: This is the type of the data of the "small" vector (typically the type of the data that will be set in the caller object).
- Caller: This is the type of the caller object, which is the object that will "invoke" the function. By default, Caller is Block.

Besides the description of a single SimpleDataMapping, we also provide a description of a vector of SimpleDataMapping. In §3.3.2.1 we present the description of a single SimpleDataMapping and in §3.3.2.2 we present the description of a vector of SimpleDataMapping.

3.3.2.1 A single SimpleDataMapping

A SimpleDataMapping is described by the following NETCDF *variables* and *dimensions*. These do not need to be in a specific NETCDF *group*, unless of course one has to specify more than one SimpleDataMapping in a *group*, in which case it is necessary to make specific *groups* because the name of a *variable*/*dimension* cannot be duplicated in a given *group*.

- The one-dimensional *variable* SetSize, an array of type netCDF::NcUint with two elements indicating the sizes (or types) of the SetFrom and SetTo sets. SetSize[0] indicates the size (or type) of the SetFrom set and SetSize[1] indicates the size (or type) of the SetTo set. For each $i \in \{0, 1\}$, if SetSize[i] == 0, then the corresponding set is a Range. Otherwise, if SetSize[i] $\neq 0$, then the corresponding set is a Subset whose size is SetSize[i]. Notice, therefore, that SetSize[i] is not the size of the corresponding set when SetSize[i] == 0. In this case, it only indicates that the set is a Range, whose size (and elements) can be determined by the SetElements *variable*. This *variable* is optional. If it is not provided, then the SetFrom and SetTo sets are assumed to be Range.

- The one-dimensional *variable* `SetElements`, of type `netCDF::NcUint`, containing the concatenation of the representations of the sets `SetFrom` and `SetTo`. A `Subset` is represented by a sequence of indices (which are the elements of the `Subset`); while a `Range` is represented by two indices a and b such that the `Range` set is given by the integers in the closed-open interval $[a, b)$. For instance, if `SetFrom` is the `Subset` $\{3, 6, 8\}$ and `SetTo` is the `Range` $[2, 5)$, then `SetElements` would be the array $(3, 6, 8, 2, 5)$.
- The *variable* `FunctionName`, whose type is `netCDF::NcString`, containing the name of the function as it is registered in the methods factory.
- The *group* `AbstractPath` containing the description of the `AbstractPath` representing the path to the caller object.

3.3.2.2 Vector of `SimpleDataMapping`

A vector of `SimpleDataMapping` is described by the following NETCDF *variables* and *dimensions*. These do not need to be in a specific NETCDF *group*, unless of course one has to specify more than one vector of `SimpleDataMapping` in a *group*, in which case it is necessary to make specific *groups* because the name of a *variable*/*dimension* cannot be duplicated in a given *group*.

- The `NumberDataMappings` *dimension*, indicating the number of `SimpleDataMappings` that is present in the vector of `SimpleDataMapping`. This *dimension* is optional. This dimension is optional. If it is not provided, then `NumberDataMappings` $= 0$ is assumed and every element in this *group* is ignored.
- The one-dimensional *variable* `DataType`, of type `netCDF::NcChar` and indexed over the `NumberDataMappings` *dimension*, specifying the type of the data that is associated with each `SimpleDataMapping` of the vector. This is the type of the data that can be set by the `SimpleDataMapping` (i.e., the `DataType` template parameter of `SimpleDataMapping`). This variable is optional. If it is not present, then the data type associated with each `SimpleDataMapping` in this vector is assumed to be `double`. If it is present then, for each $i \in \{0, \ldots,$ `NumberDataMappings` $- 1\}$, `DataType[ i ]` is the type of the data associated with the i-th `SimpleDataMapping` and can be either 'I' or 'D', indicating that the type of the data is `int` or `double`, respectively.
- The one-dimensional *variable* `SetSize`, an array of type `netCDF::NcUint` with size given by $2\times$`NumberDataMappings` indicating the size of the sets that define each `SimpleDataMapping` (the `SetFrom` and `SetTo` sets). This variable is optional. If it is not present, then all sets are assumed to be `Range`. If it is present, then `SetSize[ 2i + k ]` is the size of the `SetFrom` set of the i-th `SimpleDataMapping` if $k = 0$ or the size of the `SetTo` set of the i-th `SimpleDataMapping` if $k = 1$. If `SetSize[ j ] == 0`, then the corresponding set is a `Range`. Otherwise, the corresponding set is a `Subset` of size `SetSize[ j ]`.
- The one-dimensional *variable* `SetElements`, of type `netCDF::NcUint`, containing the concatenation of the representations of the sets `SetFrom` and `SetTo`. A `Subset` is represented by a sequence of indices (which are the elements of the `Subset`); while a `Range` is represented by two indices a and b such that the `Range` set is given by the integers in the closed-open interval $[a, b)$. If we let $\texttt{SetFrom}_i$ and $\texttt{SetTo}_i$ denote the representations of the `SetFrom` and `SetTo` sets of the i-th `SimpleDataMapping`, then `SetElements` is the array

  $$(\texttt{SetFrom}_0, \texttt{SetTo}_0, \texttt{SetFrom}_1, \texttt{SetTo}_1, \ldots, \texttt{SetFrom}_{N-1}, \texttt{SetTo}_{N-1})$$

  where $N =$ `NumberDataMappings`.
- The one-dimensional variable `FunctionName`, of type `netCDF::NcString` and indexed over `NumberDataMappings`, containing the names of the functions associated with each `SimpleDataMapping`. `FunctionName[ i ]` gives the name of the function (as registered in the methods factory) associated with the i-th `SimpleDataMapping`.
- The one-dimensional variable `Caller`, of type `netCDF::NcChar` and indexed over `NumberDataMappings`, containing the types of the caller objects associated with each `SimpleDataMapping`. `Caller[ i ]` gives the type of the caller object associated with the i-th `SimpleDataMapping` and can be either 'B', indicating that the caller is a `Block`, or 'F', indicating that the caller is a `Function`. This *variable* is optional. If it is not provided, then we assume that `Caller[ i ] = 'B'` for each $i \in \{0, \ldots,$ `NumberDataMappings` $- 1\}$, that is, we assume that all callers are `Blocks`.

- The *group* `AbstractPath`, containing a vector of `AbstractPath` with the paths to the `Block`s. The number of `AbstractPath` that this vector must contain is `NumberDataMappings`. The i-th `AbstractPath` in this vector of `AbstractPath` is the path to the `Block` associated with the i-th `SimpleDataMapping`.

4 File formats of existing `:Block`

This section collects the description of the NETCDF file formats for some the implemented `:Block` in SMS++. As such, the section will be continuously updated over the course of the project as new `:Block` will be added, although non-backward-compatible changes to existing input formats will be kept to a minimum.

4.1 `MCFBlock`

The `MCFBlock` class implements the `Block` concept for the (linear) Min-Cost Flow (MCF) problem.

The data of the problem consist of a (directed) graph $G = (N, A)$ with $n = |N|$ nodes and $m = |A|$ (directed) arcs. Each node i has a deficit $b[i]$, i.e., the amount of flow that is produced/consumed by the node: source nodes (which produce flow) have negative deficits and sink nodes (which consume flow) have positive deficits. Each arc (i, j) has an upper capacity $U[i, j]$ and a linear cost coefficient $C[i, j]$. Flow variables $X[i, j]$ represents the amount of flow to be sent on arc (i, j). Parallel arcs, i.e., multiple copies of the same arc (i, j) are in general allowed; it is expected that they have different costs (for otherwise they can be merged into a unique arc), but this is not strictly enforced. Multiple copies of some arc (i, j) can be seen as "total" flow cost on that arc being a piecewise-linear convex function. The formulation of the problem is:

$$\begin{aligned} \min\ & \textstyle\sum_{(i,j)\in A} C[i,j]X[i,j] & \\ & \textstyle\sum_{(j,i)\in A} X[j,i] - \sum_{(i,j)\in A} X[i,j] = b[i] & i \in N \\ & 0 \le X[i,j] \le U[i,j] & (i,j) \in A \end{aligned}$$

The n equations are the flow conservation constraints and the $2m$ inequalities are the flow nonnegativity and capacity constraints. At least one of the flow conservation constraints is redundant, as the demands must be balanced ($\sum_{i\in N} b[i] = 0$); indeed, exactly n - ConnectedComponents(G) flow conservation constraints are redundant, as demands must be balanced in each connected component of G.

The graph G is allowed to be "partly dynamic". The set of nodes and arcs that are input at the beginning are assumed not to be changed (save for changing costs, capacities, and deficits, and for arcs to be closed or opened). Then, new arcs and nodes, up to a set maximum, can be dynamically added or deleted. This means that the graph can be fully static (if the maximum number of dynamic arcs and nodes is set to zero) as well as fully dynamic (if the initial graph is empty).

Besides the mandatory `type` attribute of any `:Block`, the *group* representing the `MCFBlock` should contain the following:

- the *dimension* `NNodes` containing the number of nodes in the graph (n);
- the *dimension* `NArcs` containing the number of arcs in the graph (m);
- the *variable* `C`, of type `double` and indexed over the *dimension* `NArcs`; the i-th entry of the variable is assumed to contain the upper capacity of the i-th arc in the graph, that whose starting node and ending node are specified by the *variable* `SN` and `EN` (below);
- the *variable* `U`, of type `double` and indexed over the *dimension* `NArcs`; the i-th entry of the variable is assumed to contain the cost of the i-th arc in the graph (the lower capacity being fixed to 0), that whose starting node and ending node are specified by the *variable* `SN` and `EN` (below);
- the *variable* `B`, of type `double` and indexed over the *dimension* `NNodes`; the i-th entry of the variable is assumed to contain the deficit of the i-th node in the graph (note that node names here go from 0 to $n - 1$);
- the *variable* `SN`, of type `int` and indexed over the *dimension* `NArcs`; the i-th entry of the variable is assumed to contain the starting node of the i-th arc in the graph (note that node names here go from 1 to n, they are shifted by 1 w.r.t. to the indices used in the `B` variable);

- the *variable* EN, of type int and indexed over the *dimension* NArcs; the i-th entry of the variable is assumed to contain the ending node of the i-th arc in the graph (note that node names here go from 1 to n, they are shifted by 1 w.r.t. to the indices used in the B variable);
- the *dimension* DynNNodes containing the current number of dynamic nodes: all the nodes between 0 and NNodes − DynNNodes −1 are static, i.e., they cannot be deleted (and re-created), whereas all those from NNodes + DynNNode to NNodes −1 are dynamic, i.e., they can deleted and later on re-created;
- the *dimension* DynNArcs containing the current number of dynamic arcs: all the arcs between 0 and NArcs - DynNArcs −1 are static, i.e., they cannot be deleted (and re-created), whereas all those from NArcs + DynNArcs to NArcs −1 are dynamic, i.e., they can deleted and later on re-created;
- the *dimension* MaxDynNNodes containing the maximum number of dynamic nodes in the graph (the maximum value that n can take); data in the MCFBlock is allocated to accommodate for the fact that further MaxDynNN odes − DynNNodes nodes can later on be dynamically created (and deleted); if MaxDynNNodes < DynNNodes the *dimension* is ignored and NNodes is taken as the maximum overall number of nodes;
- the *dimension* MaxDynNArcs containing the maximum number of dynamic arcs in the graph (the maximum value that m can take): data in the MCFBlock is allocated to accommodate for the fact that further MaxDyn NArcs − DynNArcs arcs can later on be dynamically created (and deleted); if MaxDynNArcs < DynNArcs the *dimension* is ignored and NArcs is taken as the maximum overall number of arcs.

The two *dimensions* NNodes and NArcs are mandatory, such as are the two *variables* SN and EN. The three other *variables* are optional. If C is missing, all arc costs are assumed to be 0. If U is missing, all arc capacities are assumed to be infinite. If B is missing, all node deficits are assumed to be 0. Finally, all the *dimensions* DynNNodes, DynNArcs, MaxDynNNodes and MaxDynNArcs are optional: if they are missing they are treated as being 0 (this happening for all four means that the graph is "fully static" and cannot be changed).

4.2 UCBlock

The class UCBlock, implements the Block concept [see Block.h] for the Unit Commitment (UC) problem in electrical power production. This is typically a short-term (across for instance one week or one day time horizon) **deterministic** problem regarding finding an optimal schedule of the production of electrical generators satisfying a (large) set of technical constraints.

The model is quite flexible due to the fact that different types of units and network constraints can be used by means of the fact that the class manages son Block of type UnitBlock and NetworkBlock. Also UCBlock handles a reasonably large variety of constraints, regarding not only active power but also primary and secondary reserve and inertia. Admittedly, some choices in UCBlock are quite specific, but all the nonstandard aspects of UC can be switched away from the model (by simply not providing the data describing them). The main elements that UCBlock handles are:

- The time horizon of the problem, i.e., a discrete set of (typically, equally-spaced) time instants at which decisions are made (like, the 24 hours in a day);
- A set of electricity generating units, represented by derived classes of the base class UnitBlock;
- A set of NetworkBlock, one for each time instant in the time horizon, which represent the constraints on the electricity demand satisfaction and the technical constraints on the transmission network. These can be basically empty if the capacity of the transmission network is such as to never really impact generation decisions (a bus);

Besides the mandatory type attribute of any :Block, the *group* should contain the following:

- the *dimension* TimeHorizon containing the number of time steps in the problem;
- the *dimension* NumberUnits containing the number of units (UnitBlock) in the problem;
- the *groups* UnitBlock_0, UnitBlock_1, ..., UnitBlock_n with n == NumberUnits - 1, containing each one UnitBlock corresponding to one or more electricity generating unit (electrical generator). It is an error if the corresponding groups are not there. Each UnitBlock can have more than one electrical generator (cf.

UnitBlock::get_number_generators()), a value that is useful in the following (cf. GeneratorNode) is the total number of those, which we will refer to as NumberElectricalGenerators. This is computed by just calling get_number_generators() on each of the UnitBlock and summing all the results. Clearly, NumberElectricalGenerators >= NumberUnits and indeed, most of the UnitBlock can be expected to have just one electrical generator. If this happens for all the units then the NumberElectricalGenerators == NumberUnits. If, instead, some UnitBlock (like cascades of hydro generators or combined cycle plants) actually has more than one electrical generator, then NumberElectricalGenerators > NumberUnits (each UnitBlock must have at least one). This is because some UnitBlock (like cascades of hydro generators or combined cycle plants) can actually have more than one electrical generator in it (but each UnitBlock must have at least one). It is then useful (cf. GeneratorNode) to be able to assign a unique index g = 0, 1, ..., NumberElectricalGenerators - 1 to each of the electrical generators in the UCBlock. When NumberElectricalGenerators == NumberUnits, the index is the same as i = 0, 1, ..., NumberUnits - 1(there is a one-to-one correspondence between UnitBlock and electrical generators). When, instead, NumberElectricalGenerators > NumberUnits, a mapping must be defined. The mapping is the obvious one: UnitBlock have an ordering i = 0, 1, ..., NumberUnits - 1 (cf. the groups UnitBlock_0, UnitBlock_1, ..., above), and the electrical generators into each UnitBlock also have some natural ordering (corresponding to the columns of the matrices of variables, cf. e.g. UnitBlock::get_commitment()). Thus, in general the mapping is:

  - electrical generator 0 = first generator of UnitBlock_0
  - electrical generator 1 = second generator of UnitBlock_0
  - ...
  - electrical generator k-1 = k-th generator of UnitBlock_0 (k = UnitBlock_0->get_number_generators())
  - electrical generator k = first generator of UnitBlock_1
  - electrical generator k + 1 = second generator of UnitBlock_1
  - ...

  which of course boils down to g = i when each UnitBlock has exactly one electrical generator;

- optionally, the *dimensions* and variables necessary to deserialize a DCNetworkData object that describes the transmission network; see §4.6.2 for details. If that is not provided (basically, NumberNodes is not provided or it is == 1), then the transmission network is taken to have only one node (a bus). If the data of a DCNetworkData is specified, the DCNetworkData is passed to each of the NetworkBlock (see below) of the UCBlock, if any. However, if a NetworkBlock also has a DCNetworkData specified in its own group, then the DCNetworkData inside the NetworkBlock overrules that inside the UCBlock, which is ignored by that NetworkBlock;

- The *variable* ActivePowerDemand, of type netCDF::NcDouble, and indexed both over the dimensions NumberNodes and TimeHorizon. This variable is optional if

  - a NetworkBlock is defined for each time instant (see below), and
  - each of the defined NetworkBlock has the ActiveDemand variable specified in the corresponding group.

  Otherwise it is mandatory. When it is defined, the entry ActivePowerDemand[n , t] is assumed to contain the active power demand of node n of the transmission network at the given time instant t, where the first dimension NumberNodes can be read via NetworkBlock::NetworkData::get_number_nodes() from either the DCNetworkData object in UCBlock, or these in the NetworkBlock. When ActivePowerDemand is defined, and also the ActiveDemand variable is defined in the group of some NetworkBlock, then ActiveDemand overrules the value in the corresponding row of ActivePowerDemand, which is ignored.

- the *groups* NetworkBlock_0, NetworkBlock_1, ..., NetworkBlock_t with t = TimeHorizon - 1, containing each the constraints on the transmission network at time t. The NetworkBlocks are optional, but if any of them are missing, then

  - ActivePowerDemand (see above) is mandatory in UCBlock.

  - also the DCNetworkData (see above) is mandatory in UCBlock, unless the transmission network is a bus (that is, NumberNodes is not provided or it is == 1).

  In particular, when a NetworkBlock is not defined for a given time instant t then one of these happen:

  - If NumberNodes == 1 (or it is not provided) then no NetworkBlock is constructed for that time instant, and the entry ActivePowerDemand[0 , t] contains the active demand for t.
  - If NumberNodes > 1, then a DCNetworkBlock is automatically constructed for that time instant, it is provided with the DCNetworkData object (which must be present in UCBlock) and the row ActivePowerDemand[... , t] contains the active demand of each node at time instant t.

- the *variable* GeneratorNode, of type netCDF::NcUint and indexed over the set {0, ..., NumberElectricalGenerators - 1}; GeneratorNode[g] tells to which node of the transmission network, the specified electrical generator g belongs. Note that this means that different electrical generators in the same UnitBlock can belong to different nodes of the transmission network. This is justified e.g. by hydro cascade units where different turbines can be rather far apart geographically, but still linked by (long) stretches of rivers. If NumberElectricalGenerators == NumberUnits (all UnitBlock have exactly one electrical generator), then this variable is indexed over NumberUnits. If NumberNodes == 1 (say, it is not provided at all), then this variable need not be defined, since it is not loaded.

- the *dimension* NumberPrimaryZones tells how many primary spinning reserve zones are there in the problem. The *dimension* is optional, if it is not provided then it is taken to be 0, which means that no primary reserve constraints are present in the problem;

- the *variable* PrimaryZones, of type netCDF::NcUint and indexed over the *dimension* NumberNodes. The entry PrimaryZones[i] tells to which primary zone the node i belongs: if PrimaryZones[i] >= NumberPrimaryZones, this means that node i does not belong to any primary zone, and hence the corresponding electrical generators are not involved into the primary reserve constraints. If NumberPrimaryZones == 0 (say, it is not provided at all) then this variable need not be defined, since it is not loaded. If NumberPrimaryZones == 1 and this variable is not defined, then there is only one primary zone and all the nodes belong to it;

- the *variable* PrimaryDemand, of type netCDF::NcDouble and indexed both over the *dimensions* NumberPrimaryZones and TimeHorizon: entry PrimaryDemand[i , t] is assumed to contain the primary reserves requirement which are specified on the primary reserve zone i in the time t. If NumberPrimaryZones == 0 (say, it is not provided at all), then this variable need not be defined, since it is not loaded;

- the *dimension* NumberSecondaryZones tells how many secondary spinning reserve zones are there in the problem. The *dimension* is optional, if it is not provided then it is taken to be 0, which means that no secondary reserve constraints are present in the problem;

- the *variable* SecondaryZones, of type netCDF::NcUint and indexed over the *dimension* NumberNodes. The entry SecondaryZones[i] tells to which secondary zone the node i belongs: if SecondaryZones[i] >= NumberSecondaryZones, this means that node i does not belong to any secondary zone, and hence the corresponding electrical generators are not involved into the secondary reserve constraints. If NumberSecondaryZones == 0 (say, it is not provided at all) then this variable need not be defined, since it is not loaded. If NumberSecondaryZones == 1 and this variable is not defined, then there is only one secondary zone and all the nodes belong to it;

- the *variable* SecondaryDemand, of type netCDF::NcDouble and indexed both over the *dimensions* NumberSecondaryZones and TimeHorizon: entry SecondaryDemand[i , t] is assumed to contain the secondary reserves requirement which are specified on the secondary reserve zone i in the time t. If NumberSecondaryZones == 0 (say, it is not provided at all), then this variable need not be defined, since it is not loaded;

- the *dimension* NumberInertiaZones tells how many inertia constraints zones are there in the problem. The *dimension* is optional, if it is not provided then it is taken to be 0, which means that no inertia constraints are present in the problem;

- the *variable* InertiaZones, of type netCDF::NcUint and indexed over the *dimension* NumberNodes. The entry InertiaZones[n] tells to which inertia zone the node n belongs: if InertiaZones[n] >= NumberInertiaZones, this means that node n does not belong to any inertia zone, and hence the corresponding units are not involved into the inertia reserve constraints. If NumberInertiaZones == 0 (say, it is not provided at all) then this variable need not be defined, since it is not loaded. If NumberInertiaZones == 1 and this variable is not defined, then there is only one inertia zone and all the nodes belong to it;

- the *variable* InertiaDemand, of type netCDF::NcDouble and indexed both over the *dimensions* NumberInertiaZones and TimeHorizon: entry InertiaDemand[i , t] is assumed to contain the inertia reserves requirement which are specified on the inertia reserve zone i in the time t. If NumberInertiaZones == 0 (say, it is not provided at all), then this variable need not be defined, since it is not loaded;

- the *dimension* NumberPollutants containing the number of pollutants in the problem. The *dimension* is optional, if it is not provided then it is taken to be 0, which means that no pollutants constraints are present in the problem;

- the *variable* NumberPollutantZones of type netCDF::NcUint and indexed over the *dimension* NumberPollutants: the entry NumberPollutantZones[p] is assumed to contain the number of pollutant zones associated with pollutant p. If NumberPollutants == 0 (say, it is not provided) then this variable need not be defined, since it is not loaded. Then, all number of pollutant zones associated with each pollutant p is computed as: TotalNumberPollutantZones == NumberPollutantZone[0] + ...+ NumberPollutantZone[NumberPollutants - 1];

- the *variable* PollutantZones, of type netCDF::NcUint and indexed over the *dimensions* NumberPollutants and NumberNodes: the entry PollutantZones[p , n] tells to which pollutant zone associated with pollutant p the node n belongs. If PollutantZones[p , n] >= NumberPollutantZones[p], this means that node n does not belong to any pollutant zone, and hence the corresponding units are not involved into the pollutant budget constraints associated with pollutant p. If NumberPollutants == 0 (say, it is not provided) then this variable need not be defined, since it is not loaded;

- the *variable* PollutantBudget, of type netCDF::NcDouble and indexed over the set { 0 , ..., TotalNumberPollutantZones - 1 }: the entry PollutantBudget[n] for n = 0 , ..., TotalNumberPollutantZones - 1 is assumed to contain the pollutant budget (across all the time horizon) for the pair (zone of the pollutant , pollutant) corresponding to n. In another word, since the number of pollutant zones of each pollutant may not be equal with each other it is useful (to avoid having to store PollutantBudget as an irregular matrix) to be able to assign a unique index n = 0, 1, ..., TotalNumberPollutantZones - 1 to each pollutant budget of each pollutant zone. A mapping must be defined between each entry n and the pair (zone of the pollutant , pollutant). The mapping is the obvious one: UCBlock has a set of pollutants p = 0, 1, ..., NumberPollutants - 1, and each pollutant p may have several pollutant zones (see comments of variable NumberPollutantZones above). Thus, in general the mapping is:

  - n = 0 corresponds to the zone 0 of pollutant 0
  - n = 1 corresponds to the zone 1 of pollutant 0
  - ...
  - n = NumberPollutantZone[0] - 1 corresponds to the zone NumberPollutantZone[0] - 1 of pollutant 0
  - n = NumberPollutantZone[0] corresponds to the zone 0 of pollutant 1
  - n = NumberPollutantZone[0] + 1 corresponds to the zone 1 of pollutant 1
  - ...

  If NumberPollutants == 0 (say, it is not provided) then this variable need not be defined, since it is not loaded;

- the *variable* PollutantRho, of type netCDF::NcDouble and indexed over three *dimensions* which are TimeHorizon and NumberPollutants and the set {0, ..., NumberElectricalGenerators - 1} (see comments above). The first *dimension* can have size either 1 or TimeHorizon. In the former case the entry

`PollutantRho[ 0 , p , g ]` is assumed to contain the conversion factor of pollutant p due to the electrical generator g which is equal for all time instants t. Otherwise, the first *dimension* has full size `TimeHorizon` and the entry `PollutantRho[ t , p , g ]` gives the conversion factor of pollutant p due to the electrical generator g for time t. If `NumberPollutants == 0` (it is not provided) then this variable need not be defined, since it's not loaded.

4.3 UnitBlock

The `UnitBlock` class, which derives from the `Block`, defines a base class for any possible `unit` that can be attached to a `UCBlock`. A unit is in general a set of electrical generators tied together by some technical constraints, although many units actually correspond to only one generator. The base `UnitBlock` class only has very basic information that can characterize almost any different kind of unit:

- The time horizon of the problem;
- The number of generators in the unit (1 by default, see `get_number_generators()`);
- Four `boost::multi_array< ColVariable, 2 >` objects, that are used to store the information regarding:
  - the commitment of the generators in the unit;
  - the primary spinning reserve of the generators in the unit;
  - the secondary spinning reserve of the generators in the unit;
  - the active power produced by the generators in the unit;

Each of the `boost::multi_array< ColVariable, 2 >` has as first *dimension* the time horizon and as second *dimension* the number of generators (as returned `get_number_generators()`). There are two possible cases:

- if some `boost::multi_array< ColVariable, 2 >` is empty(), then the corresponding variable does not exist (for instance, the generator in the unit may not have reserve);
- otherwise, the `boost::multi_array< ColVariable, 2 >`, say M, must have `f_time_horizon` rows and `get_number_generators()` columns: each element $M[t, g]$ gives the variable for time step t of generator g.

The class also outputs some general information regarding how the active power and/or commitment status of each generator of the unit at a given time instant impact the unit's capability of satisfying inertia constraints, and the fixed consumption (if any) of each generator in the unit when it is off. Besides the mandatory `type` attribute of any `:Block`, the *group* representing the `UnitBlock` should contain the following:

- the *dimension* `TimeHorizon` containing the time horizon. The *dimension* is optional because the same information may be passed via the method `set_time_horizon()`, or directly retrieved from the father if it is a `UCBlock`; see the comments to `set_time_horizon()` for details.
- the *dimension* `NumberIntervals` that is provided to allow that all time-dependent data in the `UnitBlock` can only change at a subset of the time instants of the time interval, being therefore piecewise-constant (possibly, constant). `NumberIntervals` should therefore be <= `TimeHorizon`, with four distinct cases:
  1. 1 < `NumberIntervals` < `TimeHorizon`, which means that at some time instants, **but not all of them**, the values of some of the relevant data are changing; the intervals are then described in *variable* `ChangeIntervals`.
  2. `NumberIntervals` == 1, which means that the value of each relevant data in the `UnitBlock` is the same for each time instant $0, \ldots,$ `TimeHorizon` $- 1$ in the time horizon. In this case, the *variable* `ChangeIntervals` (see below) is ignored.
  3. `NumberIntervals` == `TimeHorizon`, which means that values of the relevant data changes at every time interval (in principle; of course there is nothing preventing the same value to be repeated in the NETCDF input). Also in this case the *variable* `ChangeIntervals` is ignored, since it is useless.
  4. The dimension `NumberIntervals` is not provided, which means that the values of the relevant data may be the same for each time instant (as in case 2 above) or indexed over `TimeHorizon` (as in case 3 above). Also in this case, of course, `ChangeIntervals` (see below) is ignored, and therefore it can (and should) not be present.

Note that this (together with NumberIntervals, if defined) obviously sets the maximum frequency at which data can change; if some data changes less frequently (say, it is constant), then the same value will have to be repeated. Individual data can also have specific provisions for the case where the data is all equal despite NumberIntervals saying differently.

- the *variable* ChangeIntervals, of type integer and indexed over the *dimension* NumberIntervals. The time horizon is subdivided into NumberIntervals = k of the form $[0, i_1]$,$[i_1+1, i_2]$, ... $[i_{k-1}+1, \texttt{TimeHorizon}-1]$; ChangeIntervals then has to contain $[i_1, i_2, \ldots i_{k-1}]$. Note that, therefore, ChangeIntervals has one significant value less than NumberIntervals, which means that ChangeIntervals[NumberIntervals - 1] is ignored. Anyway, the whole variable is ignored if either NumberIntervals <= 1 (such as if it is not defined), or NumberIntervals >= TimeHorizon.

4.3.1 ThermalUnitBlock

The ThermalUnitBlock class derives from UnitBlock and implements a reasonably standard thermal unit of a Unit Commitment Problem. That is, the class is designed in order to give mathematical formulation to describe the operation of large set of conventional power plants (such as nuclear, hard coal, gas turbine, gas, combined cycle, oil, ...) which are directly connected to the transmission grid. Besides the mandatory type attribute of any :Block, the group must contain all the data required by the base UnitBlock, as described in the comments to UnitBlock::deserialize(netCDF::NcGroup). In particular, we refer to that description for the crucial *dimensions* TimeHorizon, NumberIntervals and ChangeIntervals. The netCDF::NcGroup must then also contain:

- the *variable* MinPower, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector MnP[t] that, for each time instant t, contains the minimum active power output value of the unit for the corresponding time step. If MinPower has length 1 then MnP[t] contains the same value for all t. Otherwise, MinPower[i] is the fixed value of MnP[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. Note that it must be MnP[t] >= 0 for all t. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded;

- the *variable* MaxPower, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector MxP[t] that, for each time instant t, contains the maximum active power output value of the unit for the corresponding time step. If MaxPower has length 1 then MxP[t] contains the same value for all t. Otherwise, MaxPower[i] is the fixed value of MxP[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. Note that it must be MxP[t] >= 0 for all t. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded;

- the *variable* DeltaRampUp, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector DP[t] that, for each time instant t, contains the ramp-up value of the unit for the corresponding time step, i.e., the maximum possible increase of active power production w.r.t. the power that had been produced in time instant t - 1, if any. This variable is optional; if it is not provided then it is assumed that DP[t] == MxP[t], i.e., the unit can ramp up by an arbitrary amount, i.e., there are no ramp-up constraints. If DeltaRampUp has length 1 then DP[t] contains the same value for all t. Otherwise, DeltaRampUp[i] is the fixed value of DP[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- the *variable* DeltaRampDown, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector DM[t] that, for each time instant t, contains the

ramp-down value of the unit for the corresponding time step, i.e., the maximum possible decrease of active power production w.r.t. the power that had been produced in time instant `t - 1`, if any. This variable is optional; if it is not provided then it is assumed that `DM[ t ] == MxP[ t ]`, i.e., the unit can ramp down by an arbitrary amount, i.e., there are no ramp-down constraints. If `DeltaRampDown` has length 1 then `DM[ t ]` contains the same value for all t. Otherwise, `DeltaRampDown[ i ]` is the fixed value of `DM[ t ]` for all t in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ] = 0`. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon`, then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded.

- the *variable* `PrimaryRho`, of type `netCDF::NcDouble` and either of size 1 or indexed over the *dimension* `NumberIntervals` (if `NumberIntervals` is not provided, then this *variable* can also be indexed over `Time Horizon`). This is meant to represent the vector `PR[ t ]` that, for each time instant t, contains the maximum possible fraction of active power that can be used as primary reserve value of the unit for the corresponding time step. This variable is optional; if it is not provided then it is assumed that this unit may not be capable of producing any primary reserve, which correspond to `PR[ t ] == 0` for all t. If `PrimaryRho` has length 1 then `PR[ t ]` contains the same value for all t. Otherwise, `PrimaryRho[ i ]` is the fixed value of `PR[ t ]` for all t in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ] = 0`. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon`, then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded.

- the *variable* `SecondaryRho`, of type `netCDF::NcDouble` and either of size 1 or indexed over the *dimension* `NumberIntervals` (if `NumberIntervals` is not provided, then this *variable* can also be indexed over `Time Horizon`). This is meant to represent the vector `SR[ t ]` that, for each time instant t, contains the maximum possible fraction of active power that can be used as secondary reserve value of the unit for the corresponding time step. This variable is optional; if it is not provided then it is assumed that this unit may not be capable of producing any secondary reserve, which correspond to `SR[ t ] == 0` for all t. If `SecondaryRho` has length 1 then `SR[ t ]` contains the same value for all t. Otherwise, `SecondaryRho[ i ]` is the fixed value of `SR[ t ]` for all t in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ] = 0`. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon`, then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded.

- the *variable* `QuadTerm`, of type `netCDF::NcDouble` and either of size 1 or indexed over the *dimension* `Number Intervals` (if `NumberIntervals` is not provided, then this *variable* can also be indexed over `TimeHorizon`). This is meant to represent the vector `A[ t ]` that, for each time instant t, contains the quadratic term of power cost function of the unit for the corresponding time step. This variable is optional; if it is not provided then it is assumed that `A[ t ] == 0`, i.e., the cost of the unit is linear in the produced power. If `QuadTerm` has length 1 then `A[ t ]` contains the same value for all t. Otherwise, `QuadTerm[ i ]` is the fixed value of `A[ t ]` for all t in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ] = 0`. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon`, then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded.

- the *variable* `StartUpCost`, of type `netCDF::NcDouble` and either of size 1 or indexed over the *dimension* `NumberIntervals` (if `NumberIntervals` is not provided, then this *variable* can also be indexed over `Time Horizon`). This is meant to represent the vector `SC[ t ]` that, for each time instant t, contains the start up cost value of the unit for the corresponding time step. This variable is optional; if it is not provided then it is assumed that `SC[ t ] == 0`, i.e., this unit may not have any start up cost. If `StartUpCost` has length 1 then `SC[ t ]` contains the same value for all t. Otherwise, `StartUpCost[ i ]` is the fixed value of `SC[ t ]` for all t in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ] = 0`. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon`, then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded.

- the *variable* `LinearTerm`, of type `netCDF::NcDouble` and either of size 1 or indexed over the *dimension* `NumberIntervals` (if `NumberIntervals` is not provided, then this *variable* can also be indexed over `Time Horizon`). This is meant to represent the vector `B[ t ]` that, for each time instant t, contains the linear term of power cost function of the unit for the corresponding time step. This variable is optional; if it is not provided then it is assumed that `B[ t ] == 0`, i.e., the cost of the unit has no linear dependence on the produced power (say, only the quadratic one). If `LinearTerm` has length 1 then `B[ t ]` contains the same value for all t. Otherwise, `LinearTerm[ i ]` is the fixed value of `B[ t ]` for all t in the interval `[ ChangeIntervals[ i -`

1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- the *variable* ConstTerm, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector C[t] that, for each time instant t, contains the const term of power cost function of the unit for the corresponding time step. This variable is optional; if it is not provided then it is assumed that C[t] == 0, i.e., the cost of the unit has no fixed term, only those depending (linearly or quadratically) on the produced power. If ConstTerm has length 1 then C[t] contains the same value for all t. Otherwise, ConstTerm[i] is the fixed value of C[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- the scalar *variable* InitialPower, of type netCDF::NcDouble and not indexed over any *dimension*. This variable indicates the amount of the power that the unit was producing at time instant -1, i.e., before the start of the time horizon; this is necessary to compute the ramp-up and ramp-down constraints. Clearly, it must be that MaxPower >= InitialPower >= MinPower if the unit was "on" at time instant -1, and it must be that InitialPower == 0 if the unit was "off" at time instant -1. The on/off status of the unit is also encoded by the scalar variable InitUpDownTime: in particular, InitUpDownTime > 0 then the unit was on at time instant -1, and therefore InitialPower >= MinPower must hold, while if InitUpDownTime <= 0 then the unit was off at time instant -1, and therefore InitialPower == 0 by definition. In fact, if InitUpDownTime <= 0 then this variable need not be defined since it is not loaded.

- the scalar *variable* InitUpDownTime, of type netCDF::NcInt and not indexed over any *dimension* and indicates the initial time to generating the unit. If InitUpDownTime > 0, this means that the unit has been on for InitUpDownTime time stamps prior to time stamp 0 (the beginning of the horizon). If, instead, InitUpDownTime <= 0, this means that the unit has been off for - InitUpDownTime time stamps prior to time stamp 0; note that InitUpDownTime == 0 means that the unit has been just shut down at the end of time instant -1, i.e., the beginning of time instant 0.

- the positive scalar *variable* MinUpTime, of type netCDF::NcUint and not indexed over any *dimension*, which indicates the minimum allowed up time in this unit. This variable is optional, if it is not provided it is taken to be MinUpTime == 0, which mean that the unit can shut down in the very same time stamp in which it starts up.

- the positive scalar *variable* MinDownTime, of type netCDF::NcUint and not indexed over any *dimension*, which indicates the minimum allowed down time in this unit. This variable is optional, if it is not provided it is taken to be MinDownTime == 0, which mean that the unit can start up in the very same time stamp in which it starts up.

- the *variable* FixedConsumption, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector FC[t] that, for each time instant t, contains the fixed consumption of the power plant if it is OFF at time t. The variable is optional; if it is not defined, FC[t] == 0 for all time instants. If it has size 1, then FC[t] == FixedConsumption[0] for all t, regardless to what NumberIntervals says. Otherwise, FixedConsumption[i] is the fixed value of FC[t] for all t in the interval [ChangeIntervals[i - 1], ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0.

- the *variable* InertiaCommitment, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector IC[t] that, for each time instant t, contains the contribution that the unit can give to the inertia constraint for the sole fact that is is on (basically, the constant to be multiplied to the commitment variable) at time t. The variable is optional; if it is not defined, IC[t] == 0 for all time instants. If it has size 1, then IC[t] == InertiaCommitment[0] for all t, regardless to what NumberIntervals says. Otherwise, InertiaCommitment[i] is the fixed value of IC[t] for

all t in the interval [ChangeIntervals[i - 1], ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0.

4.3.2 HydroUnitBlock

The HydroUnitBlock class derives from UnitBlock and implements a reasonably standard hydro unit of a Unit Commitment Problem. That is, the class is designed in order to give mathematical formulation to describe the operation of large set of hydro storage. To model complex reservoir systems several technical parameters have to be considered. These are divided into reservoir-specific parameters, the hydro links connecting the reservoirs and finally the turbine/pump parameters. The values are collected within a reservoir database, a hydro-link database and a turbine/pump-database. Besides the mandatory type attribute of any :Block, the group must contain all the data required by the base UnitBlock, as described in the comments to UnitBlock::deserialize(netCDF::NcGroup). In particular, we refer to that description for the crucial *dimensions* TimeHorizon, NumberIntervals and ChangeIntervals. The netCDF::NcGroup must then also contain:

- The *dimension* NumberReservoirs containing the number of all reservoirs (or nodes) in the HydroUnitBlock. The *dimension* is optional, if it is not provided then it is taken to be == 1, which means that the (in principle) cascading hydro system is actually single hydro reservoir. Note, however, that a single reservoir can still have multiple hydro generating units (see NumberArcs below).

- The *dimension* NumberArcs containing the set of arcs (or units) connecting the reservoirs in cascading system. Each arc represents either a turbine generating electricity by converting potential energy of water going downhill, or a pump consuming electricity for moving water uphill.

- The *variable* StartArc, of type netCDF::NcUint and indexed over the *dimension* NumberArcs; the r-th entry of the variable is the starting point of the arc (a number in 0, ..., NumberReservoirs - 1). Note that arcs are oriented; that is, a positive flow along arc r (turbine) means that water is being taken away from StartArc[r] and delivered to EndArc[r] (see next), a negative flow (pump) means vice-versa. Note that reservoir names here go from 0 to NumberReservoirs - 1.

- The *variable* EndArc, of type netCDF::NcUint and indexed over the *dimension* NumberArcs; the r-th entry of the variable is the ending point of the arc; this is a number in 0, ..., NumberReservoirs. Note: this is NumberReservoirs and **not** NumberReservoirs - 1, because arcs can end in the "fake" reservoir NumberReservoirs. This indicates that water that flows along that arc "goes away from the system" and it is no longer counted, because it can no longer be used to produce electricity further down the river, or pumped back into one of its reservoirs. Indeed, there will be something like "the most downstream turbines": after water has been used there, it just goes away down some river and does not go to any other reservoir. Arcs are oriented (see above); StartArc[r] == EndArc[r] (a self-loop) is not allowed, but multiple arcs between the same pair of reservoirs are. Indeed, often the same physical equipment can be used both as a turbine and as a pump; in our model these are represented as two parallel arcs (but with different upper and lower flow capacity, see MinFlow and MaxFlow below).

- The *variable* MinFlow, of type netCDF::NcDouble and indexed over both *dimensions* NumberIntervals and NumberArcs. The first *dimension* may have either size 1 or size NumberIntervals (if NumberIntervals is not provided, then the size can also be TimeHorizon) whereas the second one always has size NumberArcs (if the variable is provided at all). This is meant to represent the matrix MinF[t , l] which, for each time instant t and each arc l, contains the minimum flow value of the unit. This variable is optional; if it is not provided then it is assumed that MinF[t , l]== 0, i.e., the minimum flow of the unit is zero. If the first *dimension* has size 1 then the entry MinF[0 , l] gives the fixed minimum flow value of the unit for all time steps and each arc l. Otherwise, MinFlow[i , l] is the fixed value of MinF[t , l] for all time t and arc l in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1]= 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* MaxFlow, of type netCDF::NcDouble and indexed over both *dimensions* NumberIntervals and NumberArcs. The first *dimension* may have either size 1 or size NumberIntervals (if NumberIntervals is not provided, then the size can also be TimeHorizon) whereas the second one always has size NumberArcs (if the variable is provided at all). This is meant to represent the matrix MaxF[t , l] which,

for each time instant t and each arc l, contains the maximum flow value of the unit. This variable is optional; if it is not provided then it is assumed that `MaxF[ t , l ]== 0`, i.e., the maximum flow of the unit is zero. If the first *dimension* has size 1 then the entry `MaxF[ 0 , l ]` gives the fixed maximum flow value of the unit for all time steps and each arc l. Otherwise, `MaxFlow[ i , l ]` is the fixed value of `MaxF[ t , l ]` for all time t and arc l in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ]= 0`. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon`, then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded.

Note: `MinFlow` and `MaxFlow` values can be either positive or negative (or zero); whenever `MinF[ t , l ] < MaxF[ t , l ] <= 0` for each t and l, the unit is considered a pump and whenever `0 <= MinF[ t , l ] < MaxF[ t , l ]`, the unit is considered a turbine. Note that an arc must **always** be the same kind for **all** instants, i.e., it is not allowed that a unit suddenly changes between a turbine and a pump, or vice-versa. This is because the flow-to-active-power function of turbines is a convex piecewise function with possibly many pieces (see `NumberPieces`, `LinearTerm`, `ConstantTerm` below) , whereas the flow-to-active-power function of a pump is a simple linear function. In other words, the "number of pieces" of a turbine is `>= 1`, whereas the "number of pieces" of a pump is necessarily equal to 1. In reality, the same equipment can sometimes be used both as a pump and as a turbine. In our model this is be accounted for by artificially splitting the unit into `two units`, a pump one and a turbine one, which must be done at the data processing stage. This causes the possible problem that at some time instant both the pump and the turbine be active, which is not possible in practice. This is unlikely to happen (because pumps consume more than turbines produce for the same amount of water, so this would be uneconomical), but should it ever happen, this occurrence is not handled in our model (which lets it happen).

- The *variable* `MinVolumetric`, of type `netCDF::NcDouble` and indexed over both *dimensions* `NumberReservoirs` and `NumberIntervals`. The first *dimension* always has size `NumberReservoirs` (if it is provided at all), whereas the second one may have size one or size `NumberIntervals` (if `NumberIntervals` is not provided, then the size can also be `TimeHorizon`). This is meant to represent the matrix `MinV[ r , t ]` which, for each reservoir r at each time instant t contains the minimum volumetric value of the unit for each reservoir and corresponding time step. It must be that `0 <= MinV[ r , t ] < MaxV[ r , t ]` for all r and t and . This variable is optional; if it's not provided then it is assumed that `MinV[ r , t ] == 0`, i.e., the minimum volumetric of the unit is zero. If the second *dimension* has size 1 then the entry `MinV[ r , 0 ]` gives the fixed minimum volumetric value of the unit for each reservoir r along all the time horizon. Otherwise, `MinVolumetric[ r , i ]` is the fixed value of `MinV[ r , t ]` for reservoir r and all t in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ] = 0`. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon`, then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded.
- The *variable* `MaxVolumetric`, of type `netCDF::NcDouble` and indexed over both *dimensions* `NumberReservoirs` and `NumberIntervals`. The first *dimension* always has size `NumberReservoirs` (if it is provided at all), whereas the second one may have size one or size `NumberIntervals` (if `NumberIntervals` is not provided, then the size can also be `TimeHorizon`). This is meant to represent the matrix `MaxV[ r , t ]` which, for each reservoir r at each time instant t contains the maximum volumetric value of the unit for each reservoir and corresponding time step. It must be that `0 <= MinV[ r , t ] < MaxV[ r , t ]` for all r and t and . This variable is optional; if it's not provided then it is assumed that `MaxV[ r , t ] == 0`, i.e., the maximum volumetric of the unit is zero. If the second *dimension* has size 1 then the entry `MaxV[ r , 0 ]` gives the fixed maximum volumetric value of the unit for each reservoir r along all the time horizon. Otherwise, `MaxVolumetric[ r , i ]` is the fixed value of `MaxV[ r , t ]` for reservoir r and all t in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ] = 0`. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon`, then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded.

Note: It may happen that `MinV[ r , t ] == MaxV[ r , t ]`, but only **in a subset of the time instants**. For instance, the user may want to fix the final value of the reservoir, for whatever reason. So, if `MinV` and `MaxV` are independent of t, then `MinV[ r ] < MaxV[ r ]` must surely happen. If, instead, they depend on t, then equality can be accepted at some instants (but not all of them). Besides the mandatory `type` attribute of any `:Block`, the group must contain all the data required by the base `UnitBlock`, as described in the comments to `UnitBlock::deserialize( netCDF::NcGroup )`. In particular, we refer to that description for the crucial

dimensions TimeHorizon, NumberIntervals and ChangeIntervals. The netCDF::NcGroup must then also contain:

- The *variable* Inflows, of type netCDF::NcDouble and indexed over both *dimensions* NumberReservoirs and TimeHorizon. This is meant to represent the matrix InF[r , t] which, for each reservoir r, contains the amount of water that "naturally" enters into reservoir r (because of rain, ice melting, non-controlled rivers flowing, and of course net of water leaving by evaporation, human consumption etc.) during the time interval t, and therefore that is available in the reservoir at the end of time step t (hence, the beginning of time step $t+1$, if any). This variable is optional; if it isn't defined, it is taken to be zero. Inflows can be either positive or negative.

- The *variable* MinPower, of type netCDF::NcDouble and indexed over both *dimensions* NumberIntervals and NumberArcs. The first *dimension* may have either size 1 or size NumberIntervals (if NumberIntervals is not provided, then the size can also be TimeHorizon), whereas the second one always has size NumberArcs (if it is provided at all). This is meant to represent the matrix MinP[t , l] which, for each time instant t at each arc l contains the minimum power value of the unit; it must be that MinP[t , l] < MaxP[t , l] for each time instant t and each arc l. This variable is optional; if it is not provided then it is assumed that MinP[t , l] == 0, i.e., the minimum power of all units is zero(which means, each unit is a turbine). If the first *dimension* has size 1 then the entry MinP[0 , l] is assumed to contain the minimum power of arc l for all time instants. Otherwise, MinPower[i , l] is the fixed value of MinP[t , l] for arc l and all time t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[-1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require NumberIntervals, which in fact is not loaded.

- The *variable* MaxPower, of type netCDF::NcDouble and indexed over both *dimensions* NumberIntervals and NumberArcs. The first *dimension* may have either size 1 or size NumberIntervals (if NumberIntervals is not provided, then the size can also be TimeHorizon), whereas the second one always has size NumberArcs (if it is provided at all). This is meant to represent the matrix MaxP[t , l] which, for each time instant t at each arc l contains the maximum power value of the unit; it must be that MinP[t , l] < MaxP[t , l] for each time instant t and each arc l. This variable is optional; if it is not provided then it is assumed that MaxP[t , l] == 0, i.e., the maximum power of all units is zero(which means, each unit is a turbine). If the first *dimension* has size 1 then the entry MaxP[0 , l] is assumed to contain the maximum power of arc l for all time instants. Otherwise, MaxPower[i , l] is the fixed value of MaxP[t , l] for arc l and all time t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[-1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require NumberIntervals, which in fact is not loaded.

- The *variable* DeltaRampUp, of type netCDF::NcDouble and indexed over both *dimensions* NumberIntervals and NumberArcs. The first *dimension* may have either size 1 or size NumberIntervals (if NumberIntervals is not provided, then the size can also be TimeHorizon), whereas the second one always has size NumberArcs (if it is provided at all). This is meant to represent the matrix DP[t , l] which contains the maximum possible increase of the flow rate at time instant t for arc l. This variable is optional; if it is not provided then it is assumed that DP[t , l] == MaxP[t , l] - MinP[t , l], i.e., all units can ramp up by an arbitrary amount, i.e., there are no ramp-up constraints. If the first *dimension* has size 1 then the entry DP[0 , l] is assumed to contain the maximum possible increase of the flow rate of arc l for all time instants. Otherwise, DeltaRampUp[i , l] is the fixed value of DP[t , l] for arc l and all time t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[-1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require NumberIntervals, which in fact is not loaded.

- The *variable* DeltaRampDown, of type netCDF::NcDouble and indexed over both *dimensions* NumberIntervals and NumberArcs. The first *dimension* may have either size 1 or size NumberIntervals (if NumberIntervals is not provided, then the size can also be TimeHorizon), whereas the second one always has size NumberArcs (if it is provided at all). This is meant to represent the matrix DM[t , l] which contains the maximum possible decrease of the flow rate at each time instant t of each arc l. This variable is optional; if it is not provided then it is assumed that DM[t , l] == MaxP[t , l] - MinP[t , l], i.e., the unit can ramp down by an arbitrary amount, i.e., there are no ramp-down constraints. If first *dimension*

has size 1 then the entry DM[0 , l] is assumed to contain the maximum possible decrease of the flow rate of arc l for all time instants. Otherwise, DeltaRampDown[i , l] is the fixed value of DM[t , l] for arc l and all time t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[-1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require NumberIntervals, which in fact is not loaded.

- The *variable* PrimaryRho, of type netCDF::NcDouble and indexed both over the *dimensions* NumberIntervals and NumberArcs. The first *dimension* may have either size 1 or size NumberIntervals (if NumberIntervals is not provided, then the size can also be TimeHorizon), whereas the second one always has size NumberArcs (if it is provided at all). This is meant to represent the matrix PR[t , l] which, for each time instant t and arc l, contains the maximum possible fraction of active power that can be used as primary reserve. This variable is optional, when it's not present then PR[t , l] == 0 for all t and l, i.e., the unit is not capable of producing any primary reserve. Note that only turbines can produce primary reserve, i.e., PR[t , l] > 0 ==> MaxP[t , l] > 0. If the first *dimension* has size 1 then the entry PR[0 , l] is assumed to contain the maximum possible fraction of active power that can be used as primary reserve by arc l for all time instants. Otherwise, PrimaryRho[i , l] is the fixed value of PR[t , l] for arc l and all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[-1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require NumberIntervals, which in fact is not loaded.

- The *variable* SecondaryRho, of type netCDF::NcDouble and indexed both over the *dimensions* NumberIntervals and NumberArcs. The first *dimension* may have either size 1 or size NumberIntervals (if NumberIntervals is not provided, then the size can also be TimeHorizon), whereas the second one always has size NumberArcs (if it is provided at all). This is meant to represent the matrix SR[t , l] which, for each time instant t and arc l, contains the maximum possible fraction of active power that can be used as secondary reserve. This variable is optional, when it's not present then SR[t , l] == 0 for all t and l, i.e., the unit is not capable of producing any secondary reserve. Note that only turbines can produce secondary reserve, i.e., SR[t , l] > 0 ==> MaxP[t , l] > 0. If the first *dimension* has size 1 then the entry SR[0 , l] is assumed to contain the maximum possible fraction of active power that can be used as secondary reserve by arc l for all time instants. Otherwise, SecondaryRho[i , l] is the fixed value of SR[t , l] for arc l and all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[-1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require NumberIntervals, which in fact is not loaded.

- The *variable* NumberPieces, of type netCDF::NcUint and indexed over the *dimension* NumberArcs. NumberPieces[l] tells how many pieces the concave flow-to-active-power function has for unit (arc) l. Note that pumps must necessarily have exactly one piece. The sum over all i of NumberPieces[i] is the total number of pieces (say, TotalNumberPieces). Clearly, TotalNumberPieces >= NumberArcs; if the flow-to-active-power function for all turbines only have one piece (those of pumps necessarily are so), then TotalNumberPieces == NumberArcs and there is no need to define this variable. If, instead, if it is defined, then should always be such that TotalNumberPieces >= NumberArcs.

- The *variable* LinearTerm, of type netCDF::NcDouble and indexed over the set {0 , ..., TotalNumberPieces - 1} (see NumberPieces). LinearTerm[h] gives the linear term a_h of the linear function $a_h f + b_h$ that defines the concave flow-to-active-power function for some unit; the total function if $F2AP(f) = \min\{a_h f + b_h, h \in H\}$ for some finite set H that depends on the individual unit. It is then necessary to be able to assign a unique index h = 0, 1, ..., TotalNumberPieces - 1 to each pair (unit , linear function $a_h f + b_h$). When TotalNumberPieces == NumberArcs, the index is the same as i = 0, 1, ..., NumberArcs - 1 (there is a one-to-one correspondence between each (unit) arc and each piece). When, instead, TotalNumberPieces > NumberArcs, a mapping must be defined. The mapping is the obvious one: each index of i = 0, 1, ..., NumberArcs - 1, corresponds with a unit (arc), and the linear functions for each unit (arc) also have some natural ordering. Thus, in general the mapping is:

  - piece 0 = first piece of unit (arc) 0
  - piece 1 = second piece of unit (arc) 0
  - ...
  - piece NumberPieces[0] - 1 = last piece of unit (arc) 0

- piece `NumberPieces[ 0 ]` = first piece of unit (arc) 1
- piece `NumberPieces[ 0 ] + 1` = second piece of unit (arc) 1
- ...

which of course boils down to `h = i` when each arc has exactly one piece.

- The *variable* `ConstantTerm`, of type `netCDF::NcDouble` and indexed over the set {`0 , ..., TotalNumberPieces - 1`}. `ConstantTerm[ h ]` gives the constant term b_h of the linear function $a_h f + b_h$ that defines the concave flow-to-active-power function for some unit; see the comments to `LinearTerm` for details.

- The *variable* `InertiaPower`, of type `netCDF::NcDouble` and indexed both over the *dimensions* `NumberIntervals` and `NumberArcs`. The first *dimension* may have either size 1 or size `NumberIntervals` (if `NumberIntervals` is not provided, then the size can also be `TimeHorizon`) whereas the second one always has size `NumberArcs` (if it is provided at all). This is meant to represent the matrix `IP[ t , l ]` which, for each time instant t and arc l, contains the contribution that the unit can give to the inertia constraint which depends on the active power that it is currently generating (basically, the constant to be multiplied to the active power variable) at time t for arc l. The variable is optional; if it is not defined, `IP[ t , l ] == 0` for each time instants t and arc l. If the first *dimension* has size 1 then the entry `IP[ 0 , l ]` is assumed to contain the inertia power value for arc l and all time instants t. Otherwise, `InertiaPower[ i , l ]` is the fixed value of `IP[ t , l ]` for all t in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ]= 0` and all l. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon` then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded.

- The *variable* `InitialFlowRate`, of type `netCDF::NcDouble` and indexed over the *dimension* `NumberArcs`. Each entry `InFR[ i ]` indicates the amount of the flow that was going along arc i at time instant -1. This is necessary to compute ramp-up and ramp-down limits (cf. `DeltaRampUp` and `DeltaRampDown`), and therefore it is useless if there are no ramp constraints on **any** unit (arc), in which case it is not loaded.

- The *variable* `InitialVolumetric`, of type `netCDF::NcDouble` and indexed over the *dimension* `NumberReservoirs`. Each entry `InV[ r ]` indicates the volumes of water in reservoir r at time instant -1.

- The negative or positive scalar *variable* `UphillFlow`, of type `netCDF::NcInt` and indexed over the *dimension* `NumberArcs`. Each entry `UpF[ l ]` indicates the uphill flow delay for each unit (arc) l; the nontrivial concept is detailed below. This variable is optional, if it is not provided it is taken to be `UpF[ l ] == 0`.

- The positive scalar *variable* `DownhillFlow`, of type `netCDF::NcUint` and indexed over the *dimension* `NumberArcs`. Each entry `DnF[ l ]` indicates the downhill flow delay for each arc (unit) l. This variable is optional, if it is not provided it is taken to be `DnF[ l ] == 0`.

The last two quantities require some comment. Let us assume that we have any arc l, with `( StartArc[ l ] = n , EndArc[ l ] = n' )`, which corresponds to (say) a turbine. This means that a positive flow along l implies that water is being taken away from n and delivered to n', passing through the turbine to produce flow, as graphically depicted below:

$$\backslash n/ >==========> [TURBINE] >==========> \backslash n'/$$

$$UpF[l] \qquad\qquad DnF[l]$$

The issue here is that the turbine can be geographically far enough from both n and n' that the water can take a long time (one or more time instant, especially if these are `short` such as 15 or 5 minutes) to reach the turbine from n and n' from the turbine.
A particular note of caution has to be mentioned regarding the fact that `UpF[ l ]` can be **negative**: this means that the water is used in the turbine **before** it goes out of reservoir n. This is counter-intuitive, but can be explained by the fact that the pipe between n and the turbine can be full, and therefore works as a `mini reservoir` in itself. When the turbine is started, a `bubble` (depression) is created uphill the turbine; this `bubbles up` the $n ==> TURBINE$ pipe until it reaches n, and it is only at that point that the water in n starts flowing away. Hence, there is a negative

temporal delay between the water starting flowing in the turbine and it starting flowing away from n. Note that if the $n ==> TURBINE$ pipe is rather empty, the delay is positive in that one have to start sending the water, which may take some time before filling the pipe and therefore starting the turbine. Of course these are all somewhat crude approximations of the true physical behavior, but they are accurate enough for this setting. Yet, the case `UpF[ l ] < 0` cannot be disregarded.

4.3.3 BatteryUnitBlock

The `BatteryUnitBlock` class derives from `UnitBlock` and implements a `reasonably standard` Battery storage, E-mobility, Centralized demand response, Distributed load management, Distributed storage, and Power to gas units in a single class at Unit commitment problem. This can be the case of actual physical batteries, either `large` (battery storage) or `small` (e-mobility, distributed storage), of methods that use some intermediate energy vector with limited local storage/production (power-to-gas units), as well as of `logical` mechanisms that allow to temporally shift production/consumption in a limited way, thereby acting like an energy storage (centralized demand response, distributed load management). `BatteryUnitBlock` provides a quite general concept of `battery` that covers different units which mostly fit the same mathematical equations pattern. For instance, a `BatteryUnitBlock` may or may not have a fixed demand (e-mobility has, other units have not) and it may or may not provide primary and secondary reserve (battery storage may do, but other units don't). Battery storage provide an additional flexibility to the system by shifting a surplus of electric energy (e.g. due to high renewable feeding) to times with high demand or lower renewable generation. The distributed battery storage can be aggregated in the energy cells or directly placed in a single node of the network. We will therefore not stress this dependency in the subsequent equations. We emphasize that potential contribution of batteries to inertia is still a subject of active research and should be considered as optional. Besides, since the transport sector is moving towards electrification, electric mobility will have a rising impact on the electricity system. First, electricity demand is growing due to a higher amount of electric vehicles that need to be charged. On the other hand, vehicles are used only a small amount of time while being charged over a much longer timespan (e.g. at night). This allows to shift the charging process in time and provide this flexibility to the overall energy system by means of an additional generator (vehicle-to-grid) or an additional load (power-to-vehicle). Two main differences between battery storages unit and other existing units in this class are:

- battery storages unit can do primary and secondary reserve, while other units cannot;
- some of the units may have a fixed demand that battery storages unit has not.

Moreover, as the considered storage cycle is small w.r.t. the EUC time horizon, distributed storage is not considered as seasonal storage. Hence, the associated mathematical description follows the same equations as the one provided for battery storages unit. The specificity of distributed storage only relies on the fact that it is connected to a distribution grid node.

- The *variable* `MinStorage`, of type `netCDF::NcDouble` and either of size 1 or indexed over the *dimension* `NumberIntervals` (if `NumberIntervals` is not provided, then this *variable* can also be indexed over `Time Horizon`). This is meant to represent the vector `MinS[ t ]` that, for each time instant t, contains the minimum storage level of the unit for the corresponding time step. If `MinStorage` has length 1 then `MinS[ t ]` contains the same value for all t. Otherwise, `MinStorage[ i ]` is the fixed value of `MinS[ t ]` for all t in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ] = 0`. Note that it must always be `0 <= MinS[ t ] < MaxS[ t ]` for all t. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon`, then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded.
- The *variable* `MaxStorage`, of type `netCDF::NcDouble` and either of size 1 or indexed over the *dimension* `NumberIntervals` (if `NumberIntervals` is not provided, then this *variable* can also be indexed over `Time Horizon`). This is meant to represent the vector `MaxS[ t ]` that, for each time instant t, contains the maximum storage level of the unit for the corresponding time step. If `MaxStorage` has length 1 then `MaxS[ t ]` contains the same value for all t. Otherwise, `MaxStorage[ i ]` is the fixed value of `MaxS[ t ]` for all t in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ] = 0`. Note that it must always be `[0 <=] MinS[ t ] < MaxS[ t ]` for all t. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon`, then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded.

- The *variable* MinPower, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector MinP[t] that, for each time instant t, contains the minimum active power output value of the unit for the corresponding time step. If MinPower has length 1 then MinP[t] contains the same value for all t. Otherwise, MinPower[i] is the fixed value of MinP[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. Note that it must be MinP[t] <= 0 for all t. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* MaxPower, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector MaxP[t] that, for each time instant t, contains the maximum active power output value of the unit for the corresponding time step. If MaxPower has length 1 then MaxP[t] contains the same value for all t. Otherwise, MaxPower[i] is the fixed value of MaxP[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. Note that it must be MinP[t] <= 0 for all t. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The scalar *variable* InitialPower, of type netCDF::NcDouble and not indexed over any *dimension*. This variable indicates the amount of the power that the unit was producing at time instant -1, i.e., before the start of the time horizon; this is necessary to compute the ramp-up and ramp-down constraints. This variable is optional; if DeltaRampUp and DeltaRampDown are not present, InitialPower should not be read, since there are no ramping constraints. If DeltaRampUp and DeltaRampDown are present but InitialPower is not provided, its value is taken to be 0.

- The *variable* MaxPrimaryPower, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector MaxPP[t] that, for each time instant t, contains the maximum active power that can be used as primary reserve of the unit for the corresponding time step. If MaxPrimaryPower has length 1 then MaxPP[t] contains the same value for all t. Otherwise, MaxPrimaryPower[i] is the fixed value of MaxPP[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. This variable is optional, if is not provided then MaxPP[t] == 0 for all t. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* MaxSecondaryPower, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector MaxSP[t] that, for each time instant t, contains the maximum active power that can be used as secondary reserve of the unit for the corresponding time step. If MaxSecondaryPower has length 1 then MaxSP[t] contains the same value for all t. Otherwise, MaxSecondaryPower[i] is the fixed value of MaxSP[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. This variable is optional, if is not provided then MaxSP[t] == 0 for all t. Note that MaxPP[t] == 0 implies MaxSP[t] == 0 (that is, if MaxPrimaryPower is not defined then neither should MaxSecondaryPower). If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* DeltaRampUp, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector DP[t] that, for each time instant t, contains the ramp-up value of the unit for the corresponding time step, i.e., the maximum possible increase of active power production w.r.t. the power that had been produced in time instant $t-1$, if any. If DeltaRampUp has length 1 then DP[t] contains the same value for all t. Otherwise, DeltaRampUp[i] is the fixed value of DP[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that

ChangeIntervals[- 1] = 0. This variable is optional; if it is not provided then it is assumed that DP[t] == MaxP[t], i.e., the unit can ramp up by an arbitrary amount, i.e., there are no ramp-up constraints. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* DeltaRampDown, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector DM[t] that, for each time instant t, contains the ramp-down value of the unit for the corresponding time step, i.e., the maximum possible decrease of active power production w.r.t. the power that had been produced in time instant $t-1$, if any. If DeltaRampDown has length 1 then DM[t] contains the same value for all t. Otherwise, DeltaRampDown[i] is the fixed value of DM[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. This variable is optional; if it is not provided then it is assumed that DM[t] == MaxP[t], i.e., the unit can ramp down an arbitrary amount, i.e., there are no ramp-down constraints. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* StoringBatteryRho, of type netCDF::NcDouble and to be either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector SBR[t] that, for each time instant t, contains the inefficiency of storing energy of the unit for the corresponding time step. This variable is optional; if it is not provided then it is assumed that SBR[t] == 1 for all t, i.e., no (significant) energy is spent just for storing it in the battery (this simplifies the model somewhat, see below). If StoringBatteryRho has length 1 then SBR[t] contains the same value for all t. Otherwise, StoringBatteryRho[i] is the fixed value of SBR[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]] with the assumption that ChangeIntervals[- 1] = 0. Note that it must be always such that SBR[t] <= 1 for all t (as SBR[t] is the amount of energy actually going in the battery for each 1 unit of input energy). If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* ExtractingBatterRho, of type netCDF::NcDouble and to be either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector EBR[t] that, for each time instant t, contains the inefficiency of extracting energy of the unit for the corresponding time step. This variable is optional; if it is not provided, then it is assumed that EBR[t] == 1 for all t, i.e., no (significant) energy is spent just for extracting it from the battery (this simplifies the model somewhat, see below). If ExtractingBatterRho has length 1 then EBR[t] contains the same value for all t. Otherwise, ExtractingBatterRho[i] is the fixed value of EBR[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]] with the assumption that ChangeIntervals[- 1] = 0. Note that it must be always such that EBR[t] >= 1 [>= SBR[t]] for all t (as EBR[t] is the amount of energy that is taken away from the battery to obtain 1 unit of output energy). If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon, then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

Note: the special case in which EBR[t] == SBR[t] == 1 for all t, i.e., no energy is spent for storing it in / retrieving it from the battery, leads to significantly simpler mathematical models. In particular, one single variable can be used to represent both storing and retrieving, rather than requiring two separate ones (unless primary and secondary reserve are allowed and/or the cost is defined, since this also requires using two), and the binary variables need not to be defined. For details, see the comments to generate_abstract_variable() and generate_abstract_constraints().

- The scalar *variable* InitialStorage, of type netCDF::NcDouble and not indexed over any *dimension*. This variable indicates the amount of the storage level that the unit was producing at time instant -1, i.e., before the start of the time horizon; this is necessary to compute the storage level connection with intake and outtake constraints.

- The *variable* Cost, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector C[t] that, for each time instant t, contains the monetary cost of

storing one unit or energy into, or extracting it from, the battery (the cost is the same in both cases) at the corresponding time step. This variable is optional; if it is not provided then it's taken to be zero. If `Cost` has length 1 then `C[ t ]` contains the same value for all t. Otherwise, `Cost[ i ]` is the fixed value of `C[ t ]` for all t in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ] = 0`. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon`, then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded.

- The *variable* `Demand`, of type `netCDF::NcDouble` and indexed over the *dimension* `TimeHorizon`: the entry `Demand[ t ]` is assumed to contain the amount of energy that must be discharged from the battery and `sent away for some other purpose` (say, driving your e-car) at time t. This variable is optional; if it isn't defined, then `Demand[ t ] == 0`. Otherwise the `Demand[ t ]` contains the demand value for each time instant t.

- The scalar *variable* `Kappa`, of type `netCDF::NcDouble`. This *variable* contains the factor that multiplies the minimum and maximum active power, maximum primary and secondary reserve, and the minimum and maximum storage levels, at each time instant. This *variable* is optional, if it is not provided it is taken to be `Kappa == 1`.

4.3.4 `IntermittentUnitBlock`

The `IntermittentUnitBlock` class derives from `UnitBlock` and implements a Intermittent Generation for a unit representing generation (be it centralized or distributed) by intermittent (= unreliable) sources in the unit commitment problem, such as wind farms, solar parks and run-of-the-river hydroelectricity. Each unit is supposed to be connected to a specific node of the clustered network (which means that the `distributed` case refers to `distributed in a small region`, where of course `small` depends on the granularity of the network description. The model relies mainly on historical data of local generation of wind and solar at each node of the grid; these data are used to develop normalized generation profiles associated with wind and solar generators. Intermittent generators are supposed to be able to contribute to primary and secondary reserves. Contribution to the system inertia concerns more specifically run of river generators. The potential contribution of solar or wind generation to inertia is still the subject of active research. Reserve requirements are specified in order to be symmetrically available to increase or decrease power injected into the grid. Besides the mandatory `type` attribute of any `:Block`, the group must contain all the data required by the base `UnitBlock`, as described in the comments to `UnitBlock::deserialize( netCDF::NcGroup )`. In particular, we refer to that description for the crucial *dimensions* `TimeHorizon`, `NumberIntervals` and `ChangeIntervals`. The `netCDF::NcGroup` must then also contain:

- The *variable* `MinPower`, of type `netCDF::NcDouble` and either of size 1 or indexed over the *dimension* `NumberIntervals` (if `NumberIntervals` is not provided, then this *variable* can also be indexed over `Time Horizon`). This is meant to represent the vector `MinP[ t ]` that, for each time instant t, contains the minimum potential production value of the unit for the corresponding time step. If `MinPower` has length 1 then `MinP[ t ]` contains the same value for all t. Otherwise, `MinPower[ i ]` is the fixed value of `MinP[ t ]` for all t in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ] = 0`. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon`, then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded. Note that it must be `MnP[ t ] >= 0` for all t.

- The *variable* `MaxPower`, of type `netCDF::NcDouble` and either of size 1 or indexed over the *dimension* `NumberIntervals` (if `NumberIntervals` is not provided, then this *variable* can also be indexed over `Time Horizon`). This is meant to represent the vector `MaxP[ t ]` that, for each time instant t, contains the maximum potential production value of the unit for the corresponding time step. If `MaxPower` has length 1 then `MaxP[ t ]` contains the same value for all t. Otherwise, `MaxPower[ i ]` is the fixed value of `MaxP[ t ]` for all t in the interval `[ ChangeIntervals[ i - 1 ] , ChangeIntervals[ i ] ]`, with the assumption that `ChangeIntervals[ - 1 ] = 0`. If `NumberIntervals <= 1` or `NumberIntervals >= TimeHorizon`, then the mapping clearly does not require `ChangeIntervals`, which in fact is not loaded. Note that it must be `MxP[ t ] >= MnP[ t ] [>= 0]` for all t. Yet, `MxP[ t ] == MnP[ t ]` is possible: it means that (at time instant t) the unit cannot be curtailed and cannot provide any reserve.

- The *variable* `InertiaPower`, of type `netCDF::NcDouble` and either of size 1 or indexed over the *dimension* `NumberIntervals` (if `NumberIntervals` is not provided, then this *variable* can also be indexed over

TimeHorizon). This is meant to represent the vector IP[t] which, for each time instant t, contains the contribution that the unit can give to the inertia constraint which depends on the active power that it is currently generating (basically, the constant to be multiplied to the active power variable) at time t for this unit. The variable is optional; if it is not defined, IP[t] == 0 for each time instants t. If it has size 1 then the entry IP[0] is assumed to contain the inertia power value for this unit and all time instants t. Otherwise, InertiaPower[i] is the fixed value of IP[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The scalar *variable* Gamma, of type netCDF::NcDouble and not indexed over any *dimension*. This *variable* is used to take into account an uncertainty on the maximal potential production. Note that it must be 0 <= Gamma <= 1; when Gamma == 0, the unit does not provide any reserve.

- The scalar *variable* Kappa, of type netCDF::NcDouble and not indexed over any *dimension*. This *variable* is used to multiply to the minimum and maximum power at each time instant t. This is optional, if it is not provided it is taken to be Kappa == 1.

4.3.5 SlackUnitBlock

The SlackUnitBlock class derives from UnitBlock and implements implements a slack unit; a (typically, fictitious) unit capable of producing (typically, a large amount of) active power and/or primary/secondary reserve and/or inertia at any time period completely independently from each other and from all other time periods, albeit at a (typically, huge) cost. Such a unit is typically added to a Unit Commitment problem to ensure that it has a (fictitious) feasible solution, which may help solution methods. At the very least such a modified UC would produce a "least unfeasible" solution which can be used to identify the parts of the system that lack capacity/resources. Besides the mandatory type attribute of any :Block, the group must contain all the data required by the base UnitBlock, as described in the comments to UnitBlock::deserialize(netCDF::NcGroup). In particular, we refer to that description for the crucial *dimensions* TimeHorizon, NumberIntervals and ChangeIntervals. The netCDF::NcGroup must then also contain:

- The *variable* MaxPower, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector MxP[t] that, for each time instant t, contains the maximum active power output value of the unit for the corresponding time step. If MaxPower has length 1 then MxP[t] contains the same value for all t. Otherwise, MaxPower[i] is the fixed value of MxP[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. This *variable* is optional, if is not provided then MxP[t] == 0 for all t. Note that it must be MxP[t] >= 0 for all t. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* MaxPrimaryPower, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector MaxPP[t] that, for each time instant t, contains the maximum amount of primary reserve that the unit can produce in the corresponding time step. If MaxPrimaryPower has length 1 then MaxPP[t] contains the same value for all t. Otherwise, MaxPrimaryPower[i] is the fixed value of MaxPP[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. This *variable* is optional, if is not provided then MaxPP[t] == 0 for all t. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* MaxSecondaryPower, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector MaxSP[t] that, for each time instant t, contains the maximum amount of secondary reserve that the unit can produce in the corresponding time step. If MaxSecondaryPower has length 1 then MaxSP[t] contains the same value for all t. Otherwise, MaxSecondaryPower[i] is the fixed value of MaxSP[t] for all t in the interval [ChangeIntervals[i

- 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. This *variable* is optional, if is not provided then MaxSP[t] == 0 for all t. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* MaxInertia, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector MaxI[t] which, for each time instant t, contains the maximum "amount of inertia" (contribution that the SlackUnit can give to the inertia constraint) at time t. This *variable* is optional; if it is not defined, MaxI[t] == 0 for all time instants. If it has size 1, then MaxI[t] == MaxInertia[0] for all t, regardless to what NumberIntervals says. Otherwise, MaxInertia[i] is the fixed value of MaxI[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* ActivePowerCost, of type netCDF::NcDouble and either of sie 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector APC[t] that, for each time instant t, contains the cost of producing one unit of active power at the corresponding time step. This *variable* is optional, if it is not provided then it's taken to be zero(although this is a very strange setting, as it would typically imply that all the demand, or at least as much as possible of it, is satisfied by the fictitious SlackUnit rather than from "real" ones). If ActivePowerCost has length 1 then APC[t] contains the same value for t. Otherwise, ActivePowerCost[i] is the fixed value of APC[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* PrimaryCost, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector PC[t] that, for each time instant t, contains the cost of producing one unit of primary reserve at the corresponding time step. This *variable* is optional; if it is not provided then it's taken to be zero (but this is a very strange setting, cf. the discussion in ActivePowerCost). If PrimaryCost has length 1 then PC[t] contains the same value for t. Otherwise, PrimaryCost[i] is the fixed value of PC[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* SecondaryCost, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector SC[t] that, for each time instant t, contains the cost of producing one unit of secondary reserve at the corresponding time step. This *variable* is optional; if it is not provided then it's taken to be zero (but this is a very strange setting, cf. the discussion in ActivePowerCost). If SecondaryCost has length 1 then SC[t] contains the same value for t. Otherwise, SecondaryCost[i] is the fixed value of SC[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

- The *variable* InertiaCost, of type netCDF::NcDouble and either of size 1 or indexed over the *dimension* NumberIntervals (if NumberIntervals is not provided, then this *variable* can also be indexed over TimeHorizon). This is meant to represent the vector IC[t] that, for each time instant t, contains the cost of producing "one unit" of inertia. Since the inertia-producing variable is u[t], which is in the interval [0 , 1], the cost of u[t] is MaxI[t] * IC[t]; in other words, u[t] represents the fraction of the maximum possible amount of inertia (MaxI[t]) that can be produced at time step t. This *variable* is optional; if it is not provided then it's taken to be zero(but this is a very strange setting, cf. the discussion

in ActivePowerCost). If InertiaCost has length 1 then IC[t] contains the same value for t. Otherwise, InertiaCost[i] is the fixed value of IC[t] for all t in the interval [ChangeIntervals[i - 1] , ChangeIntervals[i]], with the assumption that ChangeIntervals[- 1] = 0. If NumberIntervals <= 1 or NumberIntervals >= TimeHorizon then the mapping clearly does not require ChangeIntervals, which in fact is not loaded.

4.4 HydroSystemUnitBlock

The class HydroSystemUnitBlock, which derives from the Block, in order to define a base class for any possible hydro unit plus the linking PolyhedralFunctionBlock to describe the future value of water function in a UCBlock. Besides the mandatory type attribute of any :Block, the group should contain the following:

- The *dimension* NumberHydroUnits containing the number of hydro units (HydroUnitBlock) in the problem.
- The *groups* HydroUnitBlock_0, HydroUnitBlock_1, ..., HydroUnitBlock_(n-1), with n == NumberHydroUnits, containing each one HydroUnitBlock.
- The *group* PolyhedralFunctionBlock which contains a PolyhedralFunctionBlock, whose PolyhedralFunction represents the future value of the water (a.k.a. Bellman values) left at the end of the time horizon in all the reservoirs of all the HydroUnitBlock of the HydroSystemUnitBlock.

The future value of water function is represented by the single PolyhedralFunction which lives inside the PolyhedralFunctionBlock. The vector of active variable of PolyhedralFunction is therefore in a one-to-one correspondence with the set of ColVariable in the HydroUnitBlock that represent the amount of water left in each reservoir at the end of the time horizon. Thus, it is necessary to specify the order of the active ColVariable of the PolyhedralFunction. Let us denote by X[0], X[1], ..., X[R - 1] the vector of active ColVariable (i.e., X[i] is the one returned by get_active_var(i) and R = get_num_active_var()). Since each HydroUnitBlock can have more than one reservoir (cf. HydroUnitBlock::get_number_reservoirs()), R is just the total number of reservoir, which is computed by just calling get_number_reservoirs() on each of the HydroUnitBlock and summing all the results. Clearly, R >= NumberHydroUnits. Some of the HydroUnitBlock may have just one reservoir; if this happens for all the hydro unit blocks (but this is not likely), then R == NumberHydroUnits. In this case the mapping is obvious: X[i] is the ColVariable that represent the amount of water left in the only reservoir of HydroUnitBlock_i at the end of the time horizon. When, instead, R > NumberHydroUnits, a mapping must be defined. The mapping is the obvious one: HydroUnitBlock have an ordering n = 0, 1, ..., NumberHydroUnits - 1 (cf. the *groups* HydroUnitBlock_0, HydroUnitBlock_1, ... above), and the reservoirs into each HydroUnitBlock also have a natural ordering. Thus, in general the mapping is:

- X[0] = ColVariable representing the amount of water left in the first reservoir of HydroUnitBlock_0 at the end of the time horizon
- X[1] = ColVariable representing the amount of water left in the second reservoir of HydroUnitBlock_0 at the end of the time horizon
- ...
- X[k] = ColVariable representing the amount of water left in the k-th reservoir of HydroUnitBlock_0 at the end of the time horizon, with k = HydroUnitBlock_0->get_number_reservoirs()
- X[k + 1] = ColVariable representing the amount of water left in the first reservoir of HydroUnitBlock_1 at the end of the time horizon
- X[k + 2] = ColVariable representing the amount of water left in the second reservoir of HydroUnitBlock_1 at the end of the time horizon
- ...

This must be the format of the data (linear inequalities) that define the PolyhedralFunction: the i-th entry of each vector is related to the future value of the water stored in the reservoir identified by the above mapping. See PolyhedralFunction::deserialize() for details about how the data must be stored in the PolyhedralFunctionBlock *group*.

4.5 PolyhedralFunctionBlock

The class PolyhedralFunctionBlock derives from AbstractBlock to define the following concept: a Block whose sole distinguishing feature is a PolyhedralFunction as objective, but which can otherwise contain any kind of "abstract" Variable and Constraint (provided these are handled by the base AbstractBlock class). The rationale for the PolyhedralFunctionBlock class is that a PolyhedralFunction is a perfectly fine object in itself, but one that several solvers cannot easily deal with in its "natural" form. However, a PolyhedralFunction can also be represented in a very natural way by means of one extra continuous ColVariable plus a finite set of (dynamic) FRowConstraint with LinearFunction inside (a.k.a., linear constraints). Thus, the main feature that PolyhedralFunctionBlock implements is the ability to "present" itself (construct an "abstract representation") as either having a FRealObjective with a PolyhedralFunction inside, or having a set of (continuous) ColVariable and (linear) RowConstraint. We call the first the "natural representation" of a PolyhedralFunctionBlock, and the latter its "linearized representation". Note that most individual changes in the PolyhedralFunction result in many changes to the "linearized representation", that are properly bunched into appropriate GroupModification. Conversely, many individual changes to the "linearized representation" cannot (or would be too complex to) be implemented in the PolyhedralFunction, because each one of them individually would lead it to end in a partly inconsistent state, and only a co-ordinated set of them (say, properly bunched into an appropriate GroupModification) would work. Hence, a number of changes to the "linearized representation" are not allowed; see the comments to the protected method guts_of_add_Modification_LR() for details. Other than that, PolyhedralFunctionBlock entirely relies on the machinery provided by AbstractBlock to handle all the rest of the Block, and therefore is subject to the limitations of that class regarding what kind of Constraint, Variable and Objective are supported. Besides the mandatory type attribute of any :Block, the group should contain the following:

- all the data necessary to describe a PolyhedralFunction; see §4.5.1 for details;
- any other data necessary to represent the "arbitrary" part of the AbstractBlock; see AbstractBlock::deserialize() for details.

4.5.1 PolyhedralFunction

The PolyhedralFunction class derives from C05Function, and defines a simple implementation of a convex (or concave) Function defined by the maximum (or minimum) of a "small" number of explicitly provided affine forms. In other words, if the PolyhedralFunction depends on a set of n ColVariable, its input data is a $m \times n$ matrix A and a $m \times 1$ vector b (with m given and "small"). Serialize a PolyhedralFunction into a netCDF::NcGroup, with the following format:

- The *dimension* PolyFunction_NumVar containing the number of columns of the A matrix, i.e., the number of active variables.
- The *dimension* PolyFunction_NumRow containing the number of rows of the A matrix. The dimension is optional, if it is not provided than 0 (no rows) is assumed.
- The *variable* PolyFunction_A, of type netCDF::NcDouble and indexed over both the *dimensions* NumRow and NumVar (in this order); it contains the(row-major) representation of the matrix A. The variable is only optional if NumRow == 0.
- The *variable* PolyFunction_b, of type netCDF::NcDouble and indexed over the *dimension* NumRow, which contains the vector b. The variable is only optional if NumRow == 0.
- The *dimension* PolyFunction_sign (actually a bool), which contains the "verse" of the PolyhedralFunction, i.e., true for a convex max-function and false for a concave min-function (encoded in the obvious way, i.e., zero for false, nonzero for true). The *variable* is optional, if it is not provided true is assumed.
- The scalar *variable* PolyFunction_lb, of type netCDF::NcDouble and not indexed over any *dimension*, which contains the global lower (if PolyFunction_sign == true, upper otherwise) bound on the value of the function over all the space. The *variable* is optional; if it is not provided it means that no finite lower (upper) bound exists, i.e., the lower (upper) bound is -inf (+inf).

4.6 NetworkBlock

The class NetworkBlock, which derives from the Block, defines the basic interface for the constraints/optimization problems which describe the behaviour of the network in a specific time instant that can optionally span a number of sub time instants, in the Unit Commitment (UC) problem, as represented in UCBlock. The base class handles only basic information: it allows to read/set the characteristic of the network, and the active power demand at the different nodes in the given time instant or for the all the sub time intants spanned. This information is actually bunched together in a small "passive" NetworkData object (no methods, just a data repository) that can be either de-serialized or passed ready-made (typically, by the UCBlock). Details of the kind of network that is implemented (EC, DC equations, AC equations, OPF, ...) are entirely demanded to derived objects. The interface between a NetworkBlock and the rest of the UC is just the matrix of node per sub time instants injection variables, which will have to satisfy the technical constraints of the network. Besides the mandatory type attribute of any :Block, the group should contain the following:

- Optionally, the *dimensions* and variables necessary to a NetworkData object, that describe the network. All that is optional, because the NetworkData object can alternatively be passed to the NetworkBlock via a call to set_NetworkData(). Note that if set_NetworkData() is called, but the representation of a NetworkData object is found in the NcGroup, then the NetworkData passed by set_NetworkData() is ignored, and a new NetworkData object is read from the NcGroup and used instead.

- The ActiveDemand, of type netCDF::NcDouble, and of size NumberNodes. If the NetworkData object description is present in the NcGroup this is the *dimension* NumberNodes, but the NetworkData object is optional and it may not be there. Thus if NumberNodes is not there and ActiveDemand is, then the NetworkData object must have been passed by set_NetworkData(), and the number of nodes can be read via NetworkBlock::NetworkData::get_number_nodes(). However, ActiveDemand itself is optional. If it is not found in the NcGroup, then it **must** be passed (either before or after the call to deserialize()) by calling set_ActiveDemand(). Since both groups of data are optional, the NcGroup can actually be empty which implies that all the data will be (or have been) passed by the in-memory interface. In this case, it would clearly be preferable to **entirely avoid the NcGroup to be there**, and in fact UCBlock has provisions for the NcGroup describing the NetworkBlock to be optional (see UCBlock in §4.2).

4.6.1 DCNetworkBlock

The DCNetworkBlock class derives from NetworkBlock, and defines the standard linear constraints corresponding to the "DC model" of the transmission network in the Unit Commitment problem.

4.6.2 DCNetworkData

The DCNetworkData class is a nested sub-class which only serves to have a quick way to load all the basic data (topology and electrical characteristics) that describe the transmission network. The rationale is that while often the network does not change during the (short) time horizon of UC, it makes sense to allow for this to happen. This means that individual NetworkBlock objects may in principle have different DCNetworkData, but most often they can share the same. By bunching all the information together we make it easy for this sharing to happen. Deserialize a DCNetworkData out of a netCDF::NcGroup, which should contain the following:

- the *dimension* NumberNodes containing the number of nodes in the problem; this *dimension* is optional, if it is not provided then it is taken to be == 1; If NumberNodes == 1 (equivalently, it is not provided), the network is a "bus" formed of only one node, and therefore all the subsequent information need not to be present since it is not loaded. If NumberNodes > 1, then all the subsequent information is mandatory:

- the *dimension* NumberLines containing the number of lines in the transmission network;

- the *variable* StartLine, of type netCDF::NcUint and indexed over the *dimension* NumberLines; the i-th entry of the variable is the starting point of the line (a number in 0, ..., NumberLines - 1). Note that lines are not oriented, but the flow of energy is; that is, a positive flow along line i means that energy is being taken away from StartLine[i] and delivered to EndLine[i] (see next), a negative flow means vice-versa. Note that lines names here go from 0 to NLines.getSize() - 1;

- the *variable* EndLine, of type netCDF::NcUint and indexed over the *dimension* NumberLines; the i-th entry of the variable is the ending point of the line (a number in 0, ..., NumberLines - 1. Note that lines are not oriented, but see above). StartLine[i] == EndLine[i] (a self-loop) is not allowed, but multiple lines between the same pair of nodes are. Note that lines names here go from 0 to NLines.getSize() - 1;
- the *variable* MinPowerFlow, of type netCDF::NcDouble and indexed over the *dimension* NumberLines; the i-th entry of the variable is assumed to contain the minimum power flow on line i (note that this is typically a negative number as lines are bi-directional, see above);
- the *variable* MaxPowerFlow, of type netCDF::NcDouble and indexed over the *dimension* NumberLines; the i-th entry of the variable is assumed to contain the maximum power flow at line i (a non-negative number);
- the *variable* Susceptance, of type netCDF::NcDouble and indexed over the *dimension* NumberLines; the i-th entry of this variable is assumed to contain the susceptance of line i. Note that this is strictly a positive value.

4.6.3 ECNetworkBlock

The ECNetworkBlock class, which derives from NetworkBlock and describe the behaviour of the energy community network at a specific time instant or in a time interval, i.e., a peak period that can span an arbitrary number of sub time horizons, in the Unit Commitment problem.

4.6.4 ECNetworkData

The ECNetworkData class is a nested sub-class which only serves to have a quick way to load all the basic data that describe the community network. The rationale is that while often the network does not change during the (short) time horizon of UC, it makes sense to allow for this to happen. This means that individual NetworkBlock objects may in principle have different ECNetworkData, but most often they can share the same. By bunching all the information together we make it easy for this sharing to happen. Deserialize an ECNetworkData out of a netCDF::NcGroup, which should contain the following:

- the *dimension* NumberNodes containing the number of nodes in the problem; this *dimension* is mandatory, it cannot be == 1 since it cannot exist an energy community with just one user;
- the *dimension* NumberIntervals containing the number of intervals spanned by this current time horizon to which ECNetworkBlock refers;
- the *variable* BuyPrice, of type netCDF::NcDouble and containing the tariff that the user pays to buy electricity for all the *dimension* NumberIntervals spanned by the current time horizon;
- the *variable* SellPrice, of type netCDF::NcDouble and containing the ariff that the user gains to sell electricity for all the *dimension* NumberIntervals spanned by the current time horizon;
- the *variable* RewardPrice, of type netCDF::NcDouble and containing the tariff that the user gains when it absorbs power from the microgrid instead of from the public grid for all the *dimension* NumberIntervals spanned by the current time horizon;
- the *variable* PeakTariff, of type netCDF::NcDouble and containing the tariff that the user gains when it absorbs power from the microgrid instead of from the public grid for all the *dimension* NumberIntervals spanned by the current time horizon.

4.7 SDDPBlock

The SDDPBlock is a class that derives from Block and represents a multistage stochastic programming problem of the form

$$\min_{x_0 \in \mathcal{X}^{n_0}} f_0(x_0) + \mathbb{E}\left[\min_{x_1 \in \mathcal{X}^{n_1}} f_1(x_1) + \mathbb{E}\left[\cdots + \mathbb{E}\left[\min_{x_{T-1} \in \mathcal{X}^{n_{T-1}}} f_{T-1}(x_{T-1})\right]\right]\right],$$

where T is called the time horizon, $\mathcal{X}^{n_t} \equiv \mathcal{X}^{n_t}(x_{t-1}, \xi_t) \subseteq \mathbb{R}^{n_t}$ for each $t \in \{0, \dots, T-1\}$, and $\xi = \{\xi_t\}_{t \in \{1,\dots,T-1\}}$ is a stochastic process. Notice that x_{-1} and ξ_0 are deterministic. For each $t \in \{0, \dots, T-1\}$, we call

$$\min_{x_t \in \mathcal{X}^{n_t}} f_t(x_t) + \mathcal{V}_{t+1}(x_t, \xi_t)$$

the problem associated with stage t, where

$$\mathcal{V}_{t+1}(x_t, \xi_t) = \mathbb{E}\left[V_{t+1}(x_t, \xi_{t+1}) \mid \xi_t\right]$$

is the (expected value) cost-to-go function (also called value function, future value function, future cost function), with $\mathcal{V}_T \equiv 0$ and

$$V_t(x_{t-1}, \xi_t) = \min_{x_t \in \mathcal{X}^{n_t}} f_t(x_t) + \mathcal{V}_{t+1}(x_t, \xi_t)$$

with given x_{-1} and (deterministic) ξ_0. We consider an approximation to the problem associated with stage $t \in \{0, \dots, T-1\}$ as the problem

$$\min_{x_t \in \mathcal{X}^{n_t}} f_t(x_t) + \mathcal{P}_{t+1}(x_t) \tag{2}$$

where $\mathcal{P}_{t+1}(x_t)$ is a polyhedral function, i.e., it is a function of the form

$$\mathcal{P}_{t+1}(x_t) = \max_{i \in \{1,\dots,k_t\}} \{d_{t,i}^\top x_t + e_{t,i}\}$$

with $d_{t,i} \in \mathbb{R}^{n_t}$ and $e_{t,i} \in \mathbb{R}$ for each $i \in \{1, \dots, k_t\}$.

An `SDDPBlock` is then characterized by the following:

1. It has a time horizon T.
2. It has T sub-`Blocks`, each one being a `StochasticBlock`. The t-th sub-`Block` represents an approximation to the problem associated with stage t as defined in (2).
3. It has pointers to $T-1$ `PolyhedralFunction`. The t-th `PolyhedralFunction` represents the function $\mathcal{P}_{t+1}$ in (2) and, therefore, must be defined in the t-th sub-`Block` of this `SDDPBlock` or in any of the sub-`Blocks` of that sub-`Block`, recursively.
4. It has a set of scenarios $\mathcal{S}$. Each scenario in $\mathcal{S}$ is represented by a vector of `double` and spans all the time horizon T. Each vector is divided into T parts, each one being associated with a stage of the multistage problem. Let S denote a vector representing a scenario in $\mathcal{S}$. Then, S is defined as

   $$S = (S_0, \dots, S_{T-1})$$

   where S_t is a sub-vector of S with size s_t, for each $t \in \{0, \dots, T-1\}$, and is associated with the sub-problem at stage t, i.e., it provides data for the t-th sub-`Block` of this `SDDPBlock`. We say that S_t represents the t-th sub-scenario of the scenario represented by S.

   We assume that the sub-scenarios are organized in such a way that related random data appear in contiguous areas of the sub-scenario. For instance, suppose that the random data is associated with demand, inflow, and wind power. In this case, the data related to demand should be a contiguous sub-vector D_t of the sub-scenario associated with stage t, as well as that related to inflow (F_t) and wind power (W_t). In this example, the sub-scenario S_t could be organized as

   $$S_t = (D_t, F_t, W_t).$$

   We say that this sub-scenario has three groups of related random data. The order in which the groups of related random data appear in S_t is not relevant. We could have, for instance,

   $$S_t = (W_t, D_t, F_t).$$

   But the sub-scenario associated with stage t must respect the same order for each scenario in $\mathcal{S}$.

Besides the mandatory `type` attribute of any `Block`, an SMS++ NETCDF *group* for an `SDDPBlock` contains the following:

- The `TimeHorizon` *dimension*, containing the time horizon.

- The description of the sub-Blocks of the SDDPBlock. This is given by the sub-groups StochasticBlock and StochasticBlock_t, for each $t \in \{0, \ldots, \texttt{TimeHorizon} - 1\}$. These sub-groups are optional, but they cannot be all absent. If StochasticBlock_t is not provided for some $t \in \{0, \ldots, \texttt{TimeHorizon} - 1\}$, then the StochasticBlock *group* must be provided and contain a complete description of the t-th sub-Block of this SDDPBlock. If the StochasticBlock *group* is not provided, then StochasticBlock_t must be provided for each $t \in \{0, \ldots, \texttt{TimeHorizon} - 1\}$ and contain a complete description of the t-th sub-Block of this SDDPBlock.

  If StochasticBlock_t is provided but the description of its inner Block is not provided, then the StochasticBlock *group* must be provided and contain the description of an inner Block of a StochasticBlock. In this case, the description of the inner Block provided in the StochasticBlock *group* will be used to construct the inner Block of the StochasticBlock described by the StochasticBlock_t *group*.

  If StochasticBlock_t is provided but the description of its vector of DataMapping is not provided, then if the StochasticBlock *group* is provided and contains a description of a vector of DataMapping, then it is used to construct the vector of DataMapping of the StochasticBlock described by the StochasticBlock_t *group*.

- The AbstractPath *group* containing the description of a vector of AbstractPath as described in §3.3.1. The number of AbstractPath must be equal to either 1 or TimeHorizon-1. If the number of AbstractPath is TimeHorizon-1 then the i-th AbstractPath in this vector must be the path to the PolyhedralFunction associated with the i-th sub-Block of this SDDPBlock. The i-th AbstractPath is taken with respect to the inner Block of the i-th sub-Block of this SDDPBlock. If the number of AbstractPath in this vector is 1, then all paths are assumed to be equal: for each $i \in \{0, \ldots, \texttt{TimeHorizon} - 1\}$, the provided AbstractPath will be the path to the PolyhedralFunction associated with the i-th sub-Block of this SDDPBlock (taken with respect to the inner Block of this i-th sub-Block).

- The NumberScenarios *dimension* specifying the number of scenarios.

- The ScenarioSize *dimension* containing the size of a single scenario, which spans all stages.

- The SubScenarioSize *variable*, of type netCDF::NcUint and indexed over *dimension* TimeHorizon. This dimension is optional. If it is not provided, then all sub-scenarios are assumed to have the same size, i.e., $s_t = (\texttt{ScenarioSize}/\texttt{TimeHorizon})$ for all $t \in \{0, \ldots, \texttt{TimeHorizon}-1\}$, and ScenarioSize is a multiple of TimeHorizon. If this *dimension* is provided, then SubScenarioSize[t] is the size of the sub-scenario associated with stage t, i.e., $s_t = \texttt{SubScenarioSize[t]}$, for each $t \in \{0, \ldots, \texttt{TimeHorizon}-1\}$. In the latter case, the following must hold:

$$\texttt{ScenarioSize} = \sum_{t=0}^{\texttt{TimeHorizon}-1} \texttt{SubScenarioSize}[t].$$

- The two-dimensional *variable* Scenarios of type netCDF::NcDouble and indexed over the dimensions NumberScenarios and ScenarioSize, containing the scenarios. The i-th row of Scenarios contains the i-th scenario, so that Scenarios[i][j] is the j-th component of the i-th scenario.

- The NumberRandomDataGroups *dimension* containing the number of groups of related random data within each sub-scenario. This *dimension* is optional. If it is not provided, then we assume that there is a single group of related random data. Also, this *dimension* is meaningful only if all sub-scenarios have the same size.

- The SizeRandomDataGroups *variable*, of type netCDF::NcUint and indexed over the NumberRandomDataGroups dimension, containing the size of each group of related random data in each sub-scenario. For each $i \in \{0, \ldots, \texttt{NumberRandomDataGroups} - 1\}$, NumberRandomDataGroups[i] is the size of the i-th group of a sub-scenario. This *variable* is optional. It is required only if NumberRandomDataGroups is provided and NumberRandomDataGroups > 1.

- The StateSize *variable*, of type netCDF::NcUint and being either a scalar or a one-dimensional array indexed over TimeHorizon dimension, specifying the sizes of the states at each stage. If this *variable* is a scalar, then all states are assumed to have the same size given by StateSize. If it is an array then, for each $t \in \{0, \ldots, \texttt{TimeHorizon} - 1\}$, StateSize[t] contains the size of the state at stage t. The state being a vector, its size is the dimension of the space in which it lies.

- The `AdmissibleState` *variable*, of type `netCDF::NcDouble`, containing an admissible state for each stage. An admissible state for a stage is an state which makes the problem at that stage feasible. If `StateSize` is scalar and `AdmissibleState` has *dimension* `StateSize`, then all stages are assumed to have the same admissible state given by `AdmissibleState`. Otherwise, `AdmissibleState` contains the concatenation of the states for all stages as follows.
  - If `StateSize` is scalar then, for each $t \in \{0, \ldots, \texttt{TimeHorizon} - 1\}$, an admissible state for stage t is given by
    $$(\texttt{AdmissibleState[}s_t\texttt{]}, \ldots, \texttt{AdmissibleState[}s_t + \texttt{StateSize - 1]}),$$
    where $s_t = t \times \texttt{StateSize}$. In this case, `AdmissibleState` must have size $\texttt{TimeHorizon} \times \texttt{StateSize}$.
  - If `StateSize` is a one-dimensional array then, for each $t \in \{0, \ldots, \texttt{TimeHorizon} - 1\}$, an admissible state for stage t is given by
    $$(\texttt{AdmissibleState[}s_t\texttt{]}, \ldots, \texttt{AdmissibleState[}s_t + \texttt{StateSize[t] - 1]}),$$
    where $s_t = \sum_{i=0}^{t-1} \texttt{StateSize[i]}$. In this case, `AdmissibleState` must have size equal to
    $$\sum_{i=0}^{\texttt{TimeHorizon}-1} \texttt{StateSize[i]}.$$

4.8 StochasticBlock

Besides the mandatory `type` attribute of any `Block`, an SMS++ NETCDF *group* for a `StochasticBlock` contains the following:

- The *group* `Block`, containing the description of the inner `Block`. This *group* is optional. If it is not provided, then the inner `Block` must be provided by other means.
- The description of a vector of `SimpleDataMapping`, containing the `DataMappings` associated with this `StochasticBlock`. This is optional. For the time being, each provided `DataMapping` is expected to be a `SimpleDataMapping`. Moreover, the inner `Block` of this `StochasticBlock` will serve as the reference `Block` for both the serialization and deserialization of each `SimpleDataMapping`.

In order to better understand what the different elements, a comment about `StochasticBlock` is in order. The `StochasticBlock` class is meant to represent a `Block` whose data may be stochastic. The idea is that any `Block` could have its stochastic version without any changes in its implementation. The `StochasticBlock` comes to facilitate this process. A `StochasticBlock` is characterized by the following:

1. It has exactly one sub-`Block` (called here as the inner `Block`), which is the `Block` that is becoming stochastic. This can be any `:Block`.
2. It is aware that some data of the inner `Block` is stochastic (and may be subject to changes) and that the value for this data is represented by a vector of `double`. An instance of this vector is called a scenario for the stochastic data.
3. It has a set of `DataMapping`. This is used to both identify the stochastic data in the inner `Block` and modify the values of this data. The inner `Block` may have different types of stochastic data, located in different sub-`Blocks`. A `DataMapping` is meant to represent one piece of the stochastic data.

   To understand how it could be used, we present the following simple example. Consider a `Block` B which would be the inner `Block` of a `StochasticBlock`. This `Block` B may have all sorts of data. We could have a stochastic version of B by selecting some of its data to become uncertainties. Let us suppose that B has a vector that represents some demand over time (from time 0 to $T-1$). One could establish that the demand at times 0, 3, and 8 is stochastic. Suppose also that B has a sub-`Block` B_1 that has a vector representing costs to produce certain goods. One could also establish that the cost to produce good number 6 is stochastic. In this example, the stochastic data is formed by the demand of `Block` B at times 0, 3, and 8, and the cost to produce number 6 of `Block` B_1. `DataMapping` can then be used to identify each of those uncertainties.

We could say that a vector of length 4 is a scenario for this stochastic data such that the first three elements of this vector are values for the demands and the fourth element is a value for the cost of producing good number 6. We could use a `DataMapping` to identify that demand as stochastic data. To this end, we need two sets. One set identifies what part of the scenario (indices of that vector) is associated with the demand and the second set identifies what part of the demand of B is stochastic. In this example, the first set would be $S_1 = \{0, 1, 2\}$, which states that the values for the demand are present at positions 0, 1, and 2 of the scenario vector, and the second set would be $S_2 = \{0, 3, 8\}$, which states that the demands that are stochastic are those at positions 0, 3, and 8. We also need a way of modifying this data in B. For this, it suffices a method in B that can be used to update its demand (see `SimpleDataMapping` in §3.3.2), let us say one called `set_demand(S2, vector)`.

Another `DataMapping` could be used to identify the cost of producing good number 6 as stochastic. This `DataMapping` would have $S_1 = \{4\}$ as the first set (stating that the value for this data is at position 4 of the scenario vector) and $S_2 = \{5\}$ as the second one (indicating which good cost is stochastic). Again, we need a way to update this cost in B_1, which could be done, for instance, if B_1 provides a method like `set_cost(S2, vector)`.

4. It provides a method called `set_data()` which has a scenario (a vector) as parameter and that sets the data of the inner `Block` according to the set of `DataMapping` present in the `StochasticBlock`. In the example above, when the `set_data()` method of `StochasticBlock` is called, the values for the demands and the good cost are updated with the given values in the scenario.

   In fact, the data of the inner `Block` does not need to be stochastic in any sense. What this class provides is a means to set the value of the data of its inner `Block`.

A `StochasticBlock` should have a probability distribution (or some kind of partial stochastic process) that describes the uncertainty in it. However, for the moment, it is not supported by this class and this feature will be implemented later on. Typically, an object of this class would be used in conjunction with a scenario generator and the `set_data()` method of this object would be called to consider a particular scenario.

4.9 BendersBlock

A `BendersBlock` is a `Block` whose `Objective` is an `FRealObjective` whose `Function` is a `BendersBFunction`. Moreover, it has a vector of `ColVariable` which are the active `Variable` of that `BendersBFunction`.

Besides the mandatory `type` attribute of any `:Block`, an SMS++ NETCDF *group* for a `BendersBlock` contains the following:

- The *dimension* `NumVar` containing the number of `ColVariable` of the `BendersBlock`.
- The *group* `BendersBFunction` containing the description of the `BendersBFunction` that is the `Function` of the `FRealObjective` of the `BendersBlock`.

4.10 BendersBFunction

The `BendersBFunction` is a convenience class implementing the "abstract" concept of a Benders function of "any" `Block`. `BendersBFunction` derives from **both** `C05Function` and `Block`.

The main ingredients of a `BendersBFunction` are the following:

1. A "base" `Block` B, representing an optimization problem of the form

$$\min\{c(y) : w \leq E(y) \leq z, y \in Y\}, \tag{B}$$

   where c, w, and z are vectors of appropriate sizes, E is a function of y, and Y is a convex set. Let E_i denote the i-th component of E. If E_i is nonlinear, then it must be either convex (in which case we must have $w = -\infty$) or concave (in which case we must have $z = +\infty$). This will be the one, and only, sub-`Block` of `BendersBFunction` (when "seen" as a `Block`).

2. A matrix A with m rows and n columns, a vector b with m rows, and a vector of pairs $[(C_i, S_i)]_{i \in I}$, each pair being formed by a pointer to a `RowConstraint` of `Block` B and a `ConstraintSide`, where $I = \{1, \ldots, m\}$.

   Problem (B) would typically be associated with an original problem

$$\min\{d(x) + c(y) : g \leq Fx + E(y) \leq h, x \in X, y \in Y\} \tag{O}$$

defined in terms of variables x and y. By reformulating problem (O) as

$$\min\{d(x) + \phi(x) : x \in X\}, \tag{O'}$$

where

$$\phi(x) = \min\{c(y) : (g - Fx) \leq E(y) \leq (h - Fx), y \in Y\}, \tag{P}$$

we see that (P) assumes the form of (B) with $w = g - Fx$ and $z = h - Fx$. The `BendersBFunction` represents the function ϕ whose underlying optimization problem is given by (B). The variables x are the active `Variables` of this `BendersBFunction`. As can be seen in (P), the left- and right-hand sides of (some or all) the constraints may depend on x. The `Block` B, however, does not depend on x. In order to have the left- and right-hand sides of the constraints in (B) dependent on x, we consider the mappings $x \to g - Fx$ and $x \to h - Fx$, which are used to update the left- and right-hand sides w and z of the constraints in (B) according to the values of the variables x.

These mappings are bunched together into a single mapping M defined by the matrix A and the vector b such that

$$M(x) = Ax + b. \tag{3}$$

The i-th component of this mapping, $M_i(x) = [Ax + b]_i$, is associated with the left- or right-hand side (or both) of the `RowConstraint` of `Block` B pointed by C_i. The `ConstraintSide` S_i indicates which sides of the `RowConstraint` (pointed by) C_i are affected, as follows:

- If $S_i =$ 'L', then $M_i(x)$ gives the value of the left-hand side of the `RowConstraint` (pointed by) C_i.
- If $S_i =$ 'R', then $M_i(x)$ gives the value of the right-hand side of the `RowConstraint` (pointed by) C_i.
- If $S_i =$ 'B', then $M_i(x)$ gives the value of both the left- and right-hand side of the `RowConstraint` (pointed by) C_i. This is the case, for example, of an equality constraint, in which the left- and right-hand sides are equal.

Notice that the affected constraints must necessarily be `RowConstraint`. Moreover, if a `RowConstraint` in `Block` B has finite bounds that can be different from each other and are affected by the values of x, then it would be listed twice (once for each of its bounds). That is, there would be indices $i, j \in I$ such that $i \neq j$ and $C_i = C_j$, $S_i =$ 'L', and $S_j =$ 'R'.

> **Important:** The vector of pairs $[(C_i, S_i)]_{i \in I}$ must not contain duplicate entries (i.e., there must not exist indices $i, j \in I$ such that $i \neq j$, $C_i = C_j$ and $S_i = S_j$). Moreover, if there is $i \in I$ such that $S_i =$ 'B', there must not exist $j \in I$ such that $C_i = C_j$.

We stress that the i-th active `Variable` of this `BendersBFunction` is associated with the i-th column of the mapping matrix A. In other words, the i-th active `Variable` serves as the coefficient of the i-th column of the A matrix in the mapping M.

Note that the `BendersBFunction` is not supposed to have any `Variable` or `Constraint` (besides those defined in its only inner `Block` B).

An SMS++ NETCDF *group* for a `BendersBFunction` contains the following:

- The *dimension* `NumVar` containing the number of columns of the A matrix, i.e., the number of active variables.
- The *dimension* `NumRow` containing the number of rows of the A matrix. This *dimension* is optional; if it is not provided then 0 (no rows) is assumed.
- The *dimension* `NumNonzero`, of type `netCDF::NcUint`, containing the number of nonzero entries in the A matrix. This *dimension* is optional and determines in which format the matrix A is given. If this *dimension* is present, then `NumRow` must also be. If `NumNonzero` is not present, then the A matrix is given as a dense matrix. If it is present, then the A matrix is given in a sparse format as defined by the *variables* `Row`, `Column`, and `A`. During serialization, the following criterion is used to decide the format in which the A matrix is stored. If at most 25% of its elements are nonzero, it is stored in sparse format; otherwise, it is stored in dense format.

- The *variable* NumNonzeroAtRow, of type netCDF::NcUint and indexed over the *dimension* NumRow, containing the number of nonzero elements of the A matrix in each row. If the *dimension* NumNonzero is not present, then this *variable* is ignored. If NumNonzero is present, then NumNonzeroAtRow is optional only if NumVar, NumRow, and NumNonzero are equal. In this case, if NumNonzeroAtRow is not provided, then the A matrix is assumed to be the identity matrix. If NumVar, NumRow, and NumNonzero are not all equal, then NumNonzeroAtRow must be provided. For each $i \in \{0, \ldots, \texttt{NumRow} - 1\}$, NumNonzeroAtRow[i] is the number of nonzero elements in the i-th row of the A matrix.

- The *variable* Column, of type netCDF::NcUint and indexed over the *dimension* NumNonzero, containing the column indices of the entries of the matrix. If NumNonzero or NumNonzeroAtRow is not present, then this *variable* is ignored. For each $k \in \{0, \ldots, \texttt{NumNonzero} - 1\}$, Column[k] is the column index of the k-th nonzero entry of the A matrix, whose value is given by A[k]. Column stores the column indices in row-major order, that is, if (i, j) and (p, q) are the entries of the k-th and l-th nonzero elements of A, respectively, with $k < l$, then $i \leq p$.

- The *variable* A, of type netCDF::NcDouble. This *variable* stores the values of the elements of the A matrix of the mapping. If the *dimension* NumNonzero is provided but the *variable* NumNonzeroAtRow is not, then the A matrix is assumed to be the identity matrix and the *variable* A is ignored. If both the *dimension* NumNonzero and the *variable* NumNonzeroAtRow are provided, then the A *variable* is indexed over the NumNonzero *dimension* and contains the values of the (potentially) nonzero entries of the matrix. In this case, A[k] is the value of the k-th nonzero entry, whose column index is given by Column[k]. The nonzero elements of the matrix are given in left-to-right top-to-bottom ("row-major") order. If the *dimension* NumNonzero is not present, then A is indexed over both the NumRow and NumVar *dimensions* (in this order); it contains the (row-major) representation of the matrix A. This *variable* is optional only if NumRow == 0.

- The *variable* b, of type netCDF::NcDouble and indexed over the *dimension* NumRow, which contains the vector b. This *variable* is optional. If it is not provided, then we assume b[i] = 0 for each $i \in \{0, \ldots, \texttt{NumRow} - 1\}$.

- The *variable* ConstraintSide, of type netCDF::NcChar and indexed over the *dimension* NumRow, indicating, at position i, which side of the i-th Constraint is affected. The possible values are 'L' for the left-hand (or lower bound) side, 'R' for the right-hand (or upper bound) side, and 'B' for both sides. This *variable* is optional. If it is not provided, then all entries of this array are assumed to be 'B'.

- The *group* AbstractPath containing the description of a vector of AbstractPath to the affected RowConstraints. The number of AbstractPath in that vector must be equal to the number of rows of the A matrix and, therefore, the *dimension* associated with the number of AbstractPath is NumRow. The i-th AbstractPath in this vector must be the path to the i-th affected RowConstraint (which is associated with the i-th row of the A matrix). This *group* is optional only if NumRow == 0. Each AbstractPath is taken with respect to the inner Block of this BendersBFunction (i.e., the inner Block of this BendersBFunction must be the reference Block of the path).

- The *group* Block, containing the description of the inner Block.

- The *group* BlockConfig, containing the BlockConfig of the inner Block. This *group* is optional.

- The *group* BlockSolver, containing the BlockSolverConfig of the inner Block.

5 BlockConfig for existing :Block

As described in §3.2.1, each Block can have several Configuration parameters, which can be dealt with by a BlockConfig object. It makes therefore sense to collect here all possible Configuration parameters for existing :Block, since these may end up in a SMS++ NETCDF file.

5.1 Parameters of MCFBlock

The MCFBlock class defines the following Configuration parameters:

- `f_static_constraints_Configuration`. The static constraint of a MCF are the flow conservation equations and the bound constraints. The latter are typically implemented via a `std::vector<LB0Constraint>` with exactly m entries. However, If *all* the upper bound are $+\infty$, it is possible to skip the `std::vector< LB0Constraint >` entirely and just use the fact that the `ColVariable can be defined to be non-neg ative`. If `f_static_constraints_Configuration` is set, it is a `SimpleConfiguration<int>` whose `f_value` is not 0, the bound constraints are not implemented if it is possible to do so; otherwise they are always implemented. Note that this makes it impossible to change any upper bound.

- `f_is_feasible_Configuration`. If `f_is_feasible_Configuration` is a `SimpleConfiguration< dou ble >`, its `f_value` is taken as the parameter for deciding what "approximately feasible" exactly means while checking satisfaction of both flow conservation constraint and flow upper/lower bounds; the value is taken as the *relative* tolerance for those checks.

- `f_is_optimal_Configuration`. Checking optimality of a primal-dual solution of the MCF problems means checking that both are approximately feasible, and that they approximately satisfy the Complementary Slackness Conditions. This requires two parameters for deciding what "approximately feasible" means, one for the primal (ϵ_f) and one for the dual (ϵ_c). If `f_is_optimal_Configuration` is a `SimpleConfiguration< std::pair<double,double> >`, then $\epsilon_c =$ `f_value.first` and $\epsilon_f =$ `f_value.second`.

- `f_solution_Configuration`. For all the methods dealing with solutions, i.e., `get_Solution()` and `map_ [back/forward]_solution()`, if `f_solution_Configuration` is a `SimpleConfiguration<int>`, then its `f_value` decides what is mapped, with the following encoding: 1 means "only map the primal solution", 2 means "only map the dual solution", everything else (e.g., 0) means "map everything".

6 `ComputeConfig` for existing `:Solver`

As described in §3.2.3, each `Block` can have several `ComputeConfig` parameters, which can be dealt with by a `BlockSolverConfig` object. It makes therefore sense to collect here all possible `ComputeConfig` parameters for the existing objects that define actual values for their algorithmic parameters, since these may end up in a SMS++ NETCDF file.

6.1 Standard parameters in `Solver`

The base `Solver` class defines a set of "general" algorithmic parameters, described in the following table.

name	type	description
`intMaxIter`	`int`	The algorithmic parameter for setting the maximum number of iterations that the next call to `solve()` is allowed to execute for trying to solve the `Block`. The concept of "what exactly an iteration is" is clearly `Solver`-dependent, and the user of the `Solver` need supposedly be aware of which concrete `:Solver` it is actually using to be able to sensibly set this parameter; however, because most `Solver` will actually be iterative processes, it makes sense to offer support for this notion in the base class.
`intMaxSol`	`int`	The algorithmic parameter for setting the maximum number of different solutions to the `Block` that the `Solver` should attempt to obtain and store. Mathematical models can have (very) many solutions: an objective function precisely helps in selecting among them, but even that may not be enough to narrow the choice down to a single solution (multiple optima may exist, or quasi-optimal solutions may also be sought for for any number of reasons). On the other hand, a solution can be a "big" object, and storing it may be costly. It may be therefore helpful for a Solver to know beforehand how many different solutions the user would like to get. Among the reasonable values for this parameter, 1 says "I don't care of multiple solutions, give me only the best one", and 0 says "I don't care of solutions at all, just tell me if there is any, and what its value is". The default is 1.

intLogVerb	int	An integer parameter dictating how "verbose" the log of the Solver has to be. The specific meaning of each value is Solver-dependent, but it is intended that 0 means "no log at all", and increasing values correspond to increasing verbosity. The default value is 0 (no log).
dblMaxTime	dbl	The algorithmic parameter for setting the maximum time limit that the next call to solve() can expend in trying to solve the Block. The value is assumed to be in seconds, and it's a double (so both very quick and very slow solvers are supported). The base Solver class does not explicitly distinguish between "wall-clock time" and "CPU time", which may be rather different especially in a parallel environment, but this concept can be easily added by derived classes.
dblRelAcc	dbl	The algorithmic parameter for setting the *relative* accuracy required to the solution of the Block. This is geared towards single-objective optimization problems, and it is defined as follows: a solution for a *minimization* problem has a relative accuracy of ε if a feasible (to within the required tolerance for each Block) solution has been obtained, an upper bound ub on its objective function value has been found, a lower bound lb on the optimal value of the problem has been found, and $ub - lb \leq \varepsilon \max(|lb|, 1)$. The roles of ub and lb are suitably reversed for a maximization problem. The default is 1e-6.
dblAbsAcc	dbl	The algorithmic parameter for setting the *absolute* accuracy required to the solution of the model. This is geared towards single-objective optimization problems, and it is defined as follows: a solution for a *minimization* problem has a absolute accuracy of ε if a feasible (to within the required tolerance for each Block) solution has been obtained, an upper bound ub on its objective function value has been found, a lower bound lb on the optimal value of the problem has been found, and $ub - lb \leq \varepsilon$. The roles of ub and lb are suitably reversed for a maximization problem. The default is $+\infty$, which is intended to mean that the only working accuracy is the relative one.
dblUpCutOff	dbl	The algorithmic parameter for setting the *upper cut off* of the solution process. This is a value basically telling the Solver "when enough is enough". In particular, if the Solver obtains a certified lower bound lb on the optimal value such that $lb \geq \varepsilon$ with the provided parameter ε, then it can stop. This is a certificate that the optimal value is at least ε. For a maximization problem the condition says: I actually needed a solution with objective function value at least as good as (i.e., larger than) ε, now that I have found one the problem is as good as solved to me. Hence this parameter is analogous to the maximum absolute accuracy dblAbsAcc, but "weaker" in that it does not not require any upper bound to work. For a minimization problem, instead, the condition says: I actually needed a solution with objective function value at least as good as (i.e., smaller than) ε, now that I have know for sure that this is never going to happen the problem is as good as unfeasible to me. The default is $+\infty$.
dblLwCutOff	dbl	The algorithmic parameter for setting the *lower cut off* of the solution process. This is a value basically telling the Solver "when enough is enough". In particular, if the Solver obtains a certified upper bound ub on the optimal value such that $ub \leq \varepsilon$ with the provided parameter ε, then it can stop. This is a certificate that the optimal value is at most ε. For a minimization problem the condition says: I actually needed a solution with objective function value at least as good as (i.e., smaller than) ε, now that I have found one the problem is as good as solved to me. Hence this parameter is analogous to the maximum absolute accuracy dblAbsAcc, but "weaker" in that it does not not require any lower bound to work. For a maximization problem, instead, the condition says: I actually needed a solution with objective function value at least as good as (i.e., larger than) ε, now that I have know for sure that this is never going to happen the problem is as good as unfeasible to me. The default is $-\infty$.

dblRAccSol	dbl	The algorithmic parameter for setting the relative accuracy of the accepted solutions. It instructs the Solver not to even consider a solution among the ones to be reported intMaxSol if its objective function value is "too" bad. For a minimization problem, the objective function value of a feasible solution provides an upper bound ub on the optimal value. Assuming a lower bound lb on the optimal value of the problem has been found, a solution is deemed acceptable with the provided parameter ε if $ub - lb \leq \varepsilon \max(|ub|, |lb|, 1)$. Note that if no lb is available, the above formula can be replaced with $ub - f_{best} \leq \varepsilon \max(|ub|, |f_{best}|, 1)$, where f_{best} is the value of the best (with smallest objective value) solution found so far. The roles of ub and lb are suitably reversed for a maximization problem. The default is $+\infty$.
dblAAccSol	dbl	Similar to dblRAccSol but for an *absolute* accuracy; that is, a solution is deemed acceptable with the provided parameter ε if $ub - lb \leq \varepsilon$ or $ub - f_{best} \leq \varepsilon$ with the same notation as in dblRAccSol and the same provisions about the case of a maximization problem. The default is $+\infty$.
dblFAccSol	dbl	The algorithmic parameter for setting the maximum relative allowed violation of constraints. Whenever the Solver is incapable of finding feasible solutions (maybe because there is none), it may still be useful that it returns the "least unfeasible" ones. This parameter instructs the Solver not to even consider a solution among the ones to be reported (see intMaxSol) if its violation is "too" bad. The actual meaning of this parameter may be Solver-dependent, but it can be thought to work as the "relative constraint violation". A setting of 0 may be taken as a way to tell the Solver not to bother to produce unfeasible solutions at all, which is why this is the default value of the parameter.

6.2 Standard parameters in `CDASolver`

The CDASolver class extends the parameters of Solver with the ones described in the following table.

name	type	description
intMaxDSol	int	The algorithmic parameter for setting the maximum number of different *dual* solutions that the Solver should attempt to obtain and store. Since dual solutions can be a "big" objects (in some cases, much bigger than primal ones), storing them may be costly. It may be therefore helpful for a CDASolver to know beforehand how many different dual solutions the user would like to get. Among the reasonable values for this parameter, 1 says "I don't care of multiple dual solutions, give me only the best one", and 0 says "I don't care of dual solutions at all, just tell me if there is any, and what its value is". The default is 1.
dblRAccDSol	dbl	The algorithmic parameter for setting the relative accuracy of the accepted *dual* solutions. It instructs the CDASolver not to even consider a solution among the ones to be reported (see intMaxDSol) if its objective function value is "too" bad. If the original problem is a minimization one, its dual is a maximization one. Hence, the objective function value of a dual solution provides a lower bound lb on the optimal dual value (which may, or may not, be equal to that of the primal problem). Assuming an upper bound ub on the optimal value of the *dual* problem has been found (note that this can be obtained by means of the objective value of a feasible *primal* solution), a solution is deemed acceptable with the provided parameter ε if $ub - lb \leq \varepsilon \max(|ub|, |lb|, 1)$. Note that if no ub is available, the above formula can be replaced with $f_{best} - lb \leq \varepsilon \max(|f_{best}|, |lb|, 1)$ where f_{best} is the value of the best (with largest objective value) solution found so far. The roles of ub and lb are suitably reversed if the primal is a maximization problem, so that the dual is a minimization one. The default is $+\infty$.
dblAAccDSol	dbl	Similar to dblRAccDSol but for an *absolute* accuracy; that is, a dual solution is deemed acceptable with the provided parameter ε if $ub - lb \leq \varepsilon$ or $f_{best} - lb \leq \varepsilon$ with the same notation as in dblRAccDSol and the same provisions about the case of a maximization problem. The default is $+\infty$.

dblFAccDSol	dbl	The algorithmic parameter for setting the maximum relative allowed violation of *dual* constraints, assuming of course something like that exists in the specific dual that is being dealt with. Whenever the CDASolver is incapable of finding feasible dual solutions (maybe because there is none), it may still be useful that it returns the "least unfeasible" ones. This parameter instructs the CDASolver not to even consider a solution among the ones to be reported (see intMaxDSol) if its violation is "too" bad. The actual meaning of this parameter is necessarily be CDASolver-dependent; intuitively, it may be thought to work as the "relative constraint violation" feas_epsilon of Block if the concept of "dual constraint" is applicable. A setting of 0 may be taken as a way to tell the CDASolver not to bother to produce unfeasible solutions at all, which is why this is the default value of the parameter.

6.3 Standard parameters in **Function**

The base Function class defines a set of "general" algorithmic parameters, described in the following table.

name	type	description
intMaxIter	int	The algorithmic parameter for setting the maximum number of iterations in the next call to compute(). The concept of "what exactly an iteration is" is clearly dependent on the Function, and it may not even make sense for all Function. However, some Function will actually be iterative processes, and therefore it makes sense to offer support for this notion in the base class. The default is $+\infty$ = no limits.
dblMaxTime	dbl	The parameter for setting the maximum time limit for the next call to compute(). The value is assumed to be in seconds, and it's a double (so both very fast and very slow computations are supported). The Function class (so far) does not explicitly distinguish between "wall-clock time" and "CPU time", which may be rather different especially in a parallel environment. The default is $+\infty$.
dblRelAcc	dbl	The parameter for setting the *relative* accuracy required to the function value. That is, if both an upper bound ub and a lower bound lb on the value have been found, then compute() can stop as soon as $ub - lb \leq$ dblRelAcc $\max(\|lb\|, 1)$. The default is 1e-6.
dblAbsAcc	dbl	The parameter for setting the *absolute* accuracy required to the function value. That is, if both an upper bound ub and a lower bound lb on the value have been found, then compute() can stop as soon as $ub - lb \leq$ dblRelAcc. The default is $+\infty$, which is intended to mean that the only working accuracy is the relative one.
dblUpCutOff	dbl	The parameter for setting the "upper cut off" of the computation; that is, if a lower bound lb on the value has been found, then compute() can stop as soon as $lb \geq$ dblUpCutOff. This is a certificate that the value is at least dblUpCutOff. The default is $+\infty$, i.e., no upper cut off.
dblLwCutOff	dbl	The parameter for setting the "lower cut off" of the computation; that is, if an upper bound ub on the value has been found, then compute() can stop as soon as $ub \leq$ dblLwCutOff. This is a certificate that the value is at most dblLwCutOff. The default is $-\infty$, i.e., no lower cut off.

6.4 Standard parameters in **C05Function**

The C05Function class extends the parameters of Function with the ones described in the following table.

name	type	description
intLPMaxSz	int	The algorithmic parameter for setting the size of the "local pool", that is, the maximum number of linearizations that should be stored in the local pool. The default is 1, which corresponds to the fact that the Function can only produce a single linearization at a time (for it is, say, smooth).

intGPMaxSz	int	The algorithmic parameter for setting the size of the "global pool", that is, the maximum number of linearizations that should be stored in the local pool. The default is 0, which corresponds to the fact that the Function cannot store any linearization (for it is, say, smooth and therefore there is no need to).
dblRAccLin	dbl	A linearization (g, α) computed at the point x is "accurate" if the value of the linearization coincides with the value of the function at x, i.e., $\alpha = f(x)$. In general linearizations that are not "completely accurate" can still be useful: for instance, in the Lagrangian case an ε-optimal solution to the Lagrangian problem gives rise to a valid linearization (g, α) with $\varepsilon \geq f(x) - \alpha$. This can be deemed interesting if ε is "small", but not if ε is "large". This parameter instructs the C05Function not to bother reporting (and therefore storing in the "local pool") any linearization having a relative error with $f(x)$ larger than dblRAccLin. This would generally mean $\|f(x) - \alpha| \leq$ dblRAccLin $\max(|f(x)|, |\alpha|, 1)$, except that the value $f(x)$ may not be known exactly, with only lower and/or upper bounds on it available. The actual formula therefore depends on what information is actually available: for instance, in the Lagrangian case one knows that $f(x) \geq \alpha$, and therefore typically an upper estimate $ub \geq f(x)$ is used in the formula instead of $f(x)$. The default is 0, i.e., "only perfect linearizations are allowed".
dblAAccLin	dbl	Similar to dblRAccLin but for an *absolute* accuracy; that is, a linearization is deemed acceptable if $\|f(x) - \alpha| \leq$ dblAAccLin, except that the value $f(x)$ may not be known exactly, with only lower and/or upper bounds on it available. The actual formula therefore depends on what information is actually available: for instance, in the Lagrangian case one knows that $f(x) \geq \alpha$, and therefore typically an upper estimate $ub \geq f(x)$ is used in the formula instead of $f(x)$. The default is 0, i.e., "only perfect linearizations are allowed".

6.5 Parameters of MCFSolver<MCFC>

The MCFSolver<> class derives from CDASolver, and therefore it extends the parameters of with the ones described in the following table.

name	type	description
kReopt	int	Whether the MCF algorithm (whatever it is) should re-optimize after changes in the data, as opposed to restarting from scratch.

However, MCFSolver<> is a template class, whose template parameter actually is a concrete MCF solver object derived from MCFClass. Each of these has its own algorithmic parameters, which are described in the next subsections.

6.5.1 Parameters of MCFSolver<MCFSimplex>

name	type	description
kAlgPrimal	int	If > 0 it instructs the MCFSimplex solver to use the primal (network) simplex method, otherwise the dual one.
kAlgPricing	int	Selects which pricing rule is used within the algorithm. Possible values are kDantzig $= 0$ (Dantzig's rule, most violated constraint), kFirstEligibleArc $= 1$ (first eligible arc in round-robin), and kCandidateListPivot $= 2$ (Candidate List Pivot Rule).
kNumCandList	int	Parameter to set the number of candidate list for the Candidate List Pivot method.
kHotListSize	int	Parameter to set the size of Hot List for the Candidate List Pivot method.

7 ComputeConfig for existing :Function

7.1 Parameters of BendersBFunction

The BendersBFunction class extends the parameters of C05Function with the ones described in the following table. Setting any of these parameters causes the corresponding parameter of the CDASolver of the inner Block of

the BendersBFunction to be overwritten (if the BendersBFunction has an inner Block and this Block has a CDASolver attached to it).

name	type	description
intMaxIter	int	This parameter is associated with the Solver::intMaxIter.
intLPMaxSz	int	This parameter is associated with the CDASolver::intMaxDSol parameter.
intGPMaxSz	int	The algorithmic parameter for setting the size of the "global pool", that is, the maximum number of linearizations that should be stored in the global pool.
dblMaxTime	double	This parameter is associated with the Solver::dblMaxTime.
dblRelAcc	double	This parameter is associated with the Solver::dblRelAcc.
dblAbsAcc	double	This parameter is associated with the Solver::dblAbsAcc.
dblUpCutOff	double	This parameter is associated with the Solver::dblUpCutOff.
dblLwCutOff	double	This parameter is associated with the Solver::dblLwCutOff.
dblRAccLin	double	This parameter is associated with the CDASolver::dblRAccDSol.
dblRAccLin	double	This parameter is associated with the CDASolver::dblAAccDSol.

8 Examples

8.1 Seasonal Storage Valuation

The multistage stochastic problem in the Seasonal Storage Valuation (SSV) is represented by an SDDPBlock (see §4.7). The SDDPBlock has T nested Blocks, where T is the time horizon and, therefore, we have TimeHorizon $= T$ in the specification of SDDPBlock. For each $t \in \{0, \ldots, T-1\}$, the t-th nested Block of the SDDPBlock is a StochasticBlock (see §4.8) which is associated with stage t. The TimeHorizon of an SDDPBlock could be, for instance, 52, the number of weeks of a year. Then, each sub-Block of this SDDPBlock could be associated with a particular week of the year.

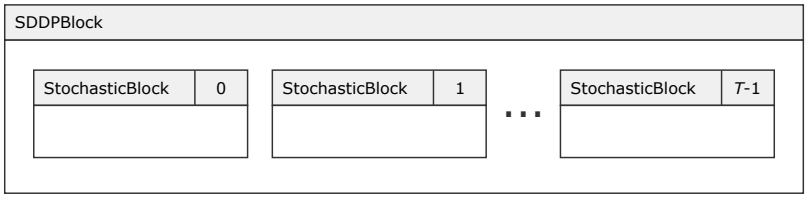


Figure 1 An SDDPBlock and its nested Blocks.

The set of scenarios in SDDPBlock is given by a matrix (see the *variable* Scenarios in §4.7), in which each row represents a scenario that spans the time horizon. If there are m scenarios (see *dimension* NumberScenarios in §4.7) and each scenario is a (row) vector in $\mathbb{R}^n$ (in which case n would be the size of a scenario; see *dimension* ScenarioSize in §4.7), a matrix $\mathcal{S} \in \mathbb{R}^{m \times n}$ with scenarios would have the form

$$\mathcal{S} = \begin{bmatrix} s_{0,0} & s_{0,1} & \cdots & s_{0,n-1} \\ s_{1,0} & s_{1,1} & \cdots & s_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ s_{m-1,0} & s_{m-1,1} & \cdots & s_{m-1,n-1} \end{bmatrix}.$$

For each $i \in \{0, \ldots, m-1\}$, the i-th scenario, represented by the vector

$$S_i = (s_{i,0}, s_{i,1}, \ldots, s_{i,n-1}),$$

is assumed to be organized in such a way that

$$S_i = (S_{i,0}, S_{i,1}, \ldots, S_{i,T-1}),$$

where $S_{i,t}$, the t-th sub-scenario of S_i, contains the data for the sub-problem at stage t (which is represented by the t-th StochasticBlock of the SDDPBlock). The sub-scenarios for different stages may have different sizes. For

each $i \in \{0, \ldots, m-1\}$ and $t \in \{0, \ldots, T-1\}$, $S_{i,t} \in \mathbb{R}^{n_t}$, so that

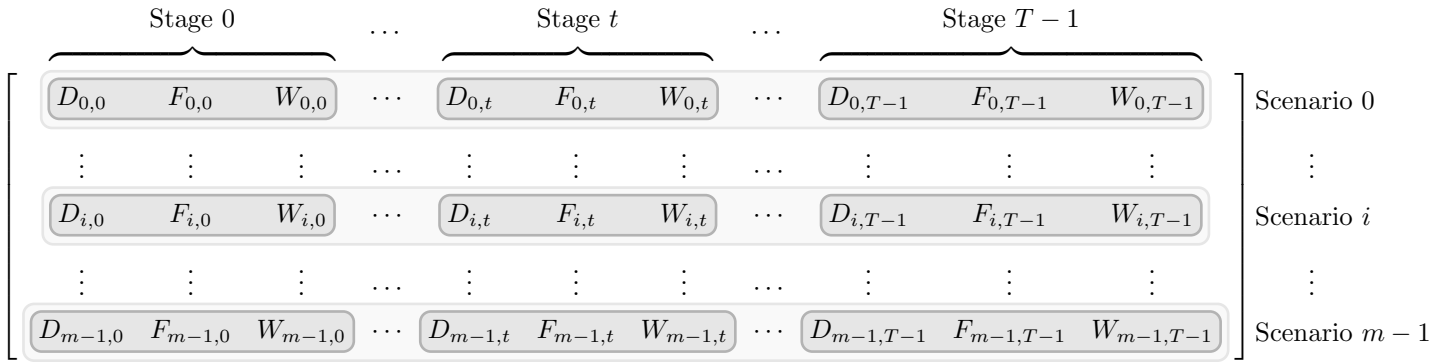
$$n = \sum_{t=0}^{T-1} n_t.$$

In the specification of `SDDPBlock`, the vector $(n_0, \ldots, n_{T-1})$ containing the sizes of the sub-scenarios is given by the *variable* `SubScenarioSize`. This *variable* is optional and, in the case it is not provided, all sub-scenarios are assumed to have the same size, i.e., $n_t = n/T$ for each $t \in \{0, \ldots, T-1\}$, and n is a multiple of T.

If all sub-scenarios have the same size, their data must be organized so that related random data are in contiguous parts of the vector representing a sub-scenario. For instance, suppose that each sub-scenario contains data related to demand, inflow, and wind power. In this example, we would say that there are three groups of data (the *dimension* `NumberRandomDataGroups` in `SDDPBlock` would then be 3) and, for each $i \in \{0, \ldots, m-1\}$ and $t \in \{0, \ldots, T-1\}$, the sub-scenario $S_{i,t}$ would be written as

$$S_{i,t} = (D_{i,t}, F_{i,t}, W_{i,t}) .$$

The *variable* `SizeRandomDataGroups` of `SDDPBlock` would then be an array containing the sizes of the sub-vectors $D_{i,t}$, $F_{i,t}$, and $W_{i,t}$. For instance, if $D_{i,t}$ is a vector with size 7, $F_{i,t}$ is one with size 4, and $W_{i,t}$ is one with size 11, the *variable* `SizeRandomDataGroups` would contain the array $(7, 4, 11)$. Hence, the matrix $\mathcal{S}$ would then have the following form:



In this example, if $T = 52$, the *dimension* `ScenarioSize` would be $52(7 + 4 + 11)$. The light gray box indicate a scenario while a dark gray box indicates a sub-scenario.

Each sub-`Block` of the `SDDPBlock` represents a short-term problem that is associated with a `UCBlock` as follows. For each $t \in \{0, \ldots, T-1\}$, the inner `Block` of the t-th `StochasticBlock` of the `SDDPBlock` is a `BendersBlock`. The `BendersBlock` has a set of `Variables` and an `Objective`. Its `Objective` is an `FRealObjective` whose `Function` is a `BendersBFunction`. The `Variables` of the `BendersBlock` are the ones that are active in the `BendersBFunction`. The inner `Block` of the `BendersBFunction` is a `UCBlock` (or a Lagrangian relaxation of a `UCBlock`) which depends on the values of the `Variables` that are active in the `BendersBFunction`. In the SSV, the `Variables` defined in the `BendersBlock` (which are active in the `BendersBFunction`) represent the initial volumes of the reservoirs. These affect the values of the left- or right-hand side of some `Constraints` of the `UCBlock`. The affected `Constraints` and how they are affected are determined by the `BendersBFunction`. In the `BendersBFunction`, the affected `Constraints` are indicated by a vector of `AbstractPath` (each path being relative to the `BendersBlock`) and the affected sides of the `Constraints` are indicated by the one-dimensional *variable* `ConstraintSide`. The way the `Constraints` are affected by those `Variables` is defined by the linear mapping given by the NETCDF *variables* `A` and `b` (together with their auxiliary *variables* and *dimensions*). Figure 2 summarizes the structure of an `StochasticBlock` in which x_t represent the `Variables` of the `BendersBlock`.

8.1.1 `BendersBlock`

Before giving an example on how to construct the `BendersBlock`, we shall look into some details of the `UCBlock`. Following the example above, in which the `SDDPBlock` represents a multistage problem over 52 weeks, there would be one `UCBlock` associated with each of those 52 weeks. Each `UCBlock` could be, for instance, a deterministic multistage problem with 168 stages (one stage for each hour of the week). In this case, the dimension `TimeHorizon` in the specification of the `UCBlock` would be 168. As you can see, there are two independent time horizons: one at the level

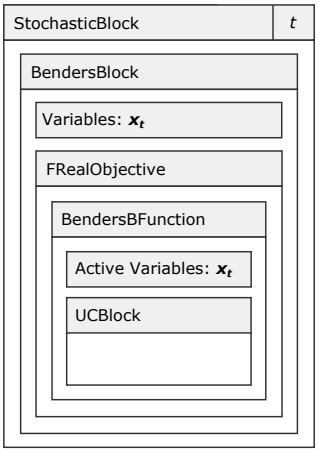


Figure 2 Structure of an `StochasticBlock` in the SSV.

of the `SDDPBlock` (the mid-term problem) and one at the level of the `UCBlock` (the short-term problem). To avoid confusion, we will refer to the time horizon and stages of the `SDDPBlock` as the outer time horizon and outer stages, respectively, and the time horizon and stages of the `UCBlock` as the inner time horizon and inner stages, respectively. We will also use the notations T^{out} and T^{in} to refer to the outer and inner time horizons when necessary.

A `UCBlock` has two sets of sub-`Blocks`. The first one is a set of `UnitBlocks` whose size is given by the `NumberUnits` *dimension*. The second one is a set of `NetworkBlocks` whose size is (inner) `TimeHorizon`. Figure 3 illustrates the `UCBlock` and its nested `Blocks`. In this figure, N is the number of `UnitBlocks` (*dimension* `NumberUnits`) and T^{in} is the inner time horizon (*dimension* `TimeHorizon` in the specification of `UCBlock`).

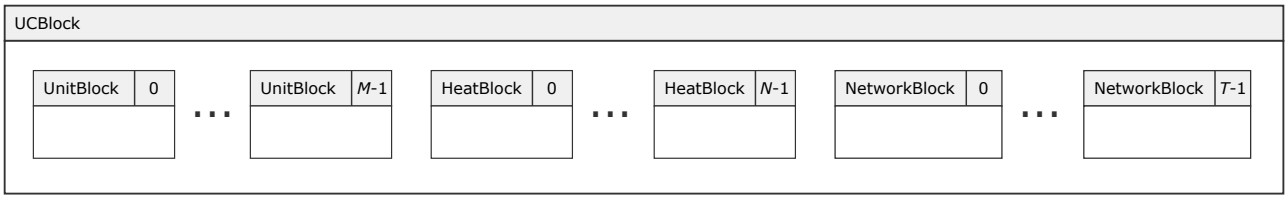


Figure 3 `UCBlock` and its nested `Blocks`.

There can be many types of `UnitBlocks` in the `UCBlock` (e.g., `ThermalUnitBlock`, `BatteryUnitBlock`, ...). In the SSV, exactly one of those `UnitBlocks` is a `HydroSystemUnitBlock`. The `UnitBlocks` of the `UCBlock` can be given in any order the user may wish. But to simplify the example, we assume that the `HydroSystemUnitBlock` is the first of the `UnitBlocks` (in Figure 3, it would be the `UnitBlock` with index 0). Hence, the description of the `HydroSystemUnitBlock` should be given in the *group* `UnitBlock_0` in the specification of the `UCBlock`.

The `HydroSystemUnitBlock` (see §4.4) is a `UnitBlock` that has a list of `HydroUnitBlocks` (see §4.3.2) and one `PolyhedralFunctionBlock` as nested `Blocks`. The `PolyhedralFunctionBlock` is the last among the nested `Blocks` of the `HydroSystemUnitBlock`. Figure 4 depicts the `HydroSystemUnitBlock` with its nested `Blocks` in which H denotes the number of `HydroUnitBlocks` (*dimension* `NumberHydroUnits` in the specification of the `HydroSystemUnitBlock`).

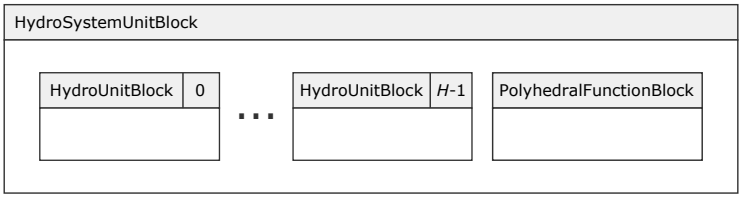


Figure 4 `HydroSystemUnitBlock` and its nested `Blocks`.

The `Variables` of the `BendersBlock` represent the dependence between consecutive `UnitBlocks`. In the SSV,

this dependence is related to the volumes of the reservoirs: the initial volume of a reservoir of a `HydroUnitBlock` at the outer stage $t \in \{1, \dots, T^{\text{out}} - 1\}$ must be equal to the volume of that reservoir at the last inner stage of the `HydroUnitBlock` at the outer stage $t - 1$. The `Variables` of the `BendersBlock` at the outer stage $t \in \{0, \dots, T^{\text{out}} - 1\}$ thus represent the initial volumes of the reservoirs of the `HydroUnitBlocks` at that stage.

Let r_i denote the number of reservoirs (*dimension* `NumberReservoirs` in the specification of `HydroUnitBlock`) of the i-th `HydroUnitBlock`. For each $i \in \{0, \dots, H-1\}$ and $j \in \mathcal{N}^i = \{0, \dots, r_i - 1\}$, let $\tilde{v}_{i,j}$ denote the initial volume of the j-th reservoir of the i-th `HydroUnitBlock`. These initial volumes appear in the "final volumes constraints" of the `HydroUnitBlock`. For the inner stage $t = 0$ and the i-th `HydroUnitBlock`, these constraints are

$$v^i_{j,t} = \tilde{v}_{i,j} + F^i_{j,t} + \Big(\sum_{l=(d,j)\in\mathcal{L}^i} f^i_{t-\tau_l^{i,\text{dn}},l} - \sum_{l=(j,d)\in\mathcal{L}^i} f^i_{t-\tau_l^{i,\text{up}},l} \Big), \quad \forall j \in \mathcal{N}^i \text{ and } t = 0. \tag{4}$$

In these constraints, $v^i_{j,t}$ denotes the volume of the j-th reservoir at the (inner) stage t, $F^i_{j,t}$ denotes the inflow at the j-th reservoir at the (inner) stage t, $f^i_{k,l}$ is the flow rate at time k for arc l, and $\tau_l^{i,\text{dn}}$ and $\tau_l^{i,\text{up}}$ denote the uphill and downhill flow rates at arc l, respectively. (See the documentation of the `HydroUnitBlock` class for details.)

Therefore, we could define the `Variables` x of the `BendersBlock` as that representing the following vector:

$$x = \Big[\underbrace{\tilde{v}_{0,0} \quad \dots \quad \tilde{v}_{0,r_0-1}}_{\texttt{HydroUnitBlock } 0} \quad \dots \quad \underbrace{\tilde{v}_{k,0} \quad \dots \quad \tilde{v}_{k,r_k-1}}_{\texttt{HydroUnitBlock } k} \quad \dots \quad \underbrace{\tilde{v}_{H-1,0} \quad \dots \quad \tilde{v}_{H-1,r_{H-1}-1}}_{\texttt{HydroUnitBlock } H-1} \Big]^\top. \tag{5}$$

As aforementioned, these are the active `Variables` of the `BendersBFunction`. Based on the values of these `Variables`, the `BendersBFunction` is responsible for updating the `Constraints` of the `HydroUnitBlocks`. By isolating the independent terms in constraints (4), we can rewrite them as

$$v^i_{j,t} - \Big(\sum_{l=(d,j)\in\mathcal{L}^i} f^i_{t-\tau_l^{i,\text{dn}},l} - \sum_{l=(j,d)\in\mathcal{L}^i} f^i_{t-\tau_l^{i,\text{up}},l} \Big) = \tilde{v}_{i,j} + F^i_{j,t}, \quad \forall j \in \mathcal{N}^i \text{ and } t = 0. \tag{6}$$

Notice that the independent terms appear in the right-hand side of this constraint. Since it is an equality constraint, the left- and right-hand sides of the `RowConstraint` representing this constraint must be equal to $\tilde{v}_{i,j} + F^i_{j,t}$ and, therefore, are identically affected by the initial volumes of the reservoir. This means that we could define the mapping M in the `BendersBFunction` (see §4.10) as

$$M(x) = x + F,$$

where

$$F = \Big[\underbrace{F^0_{0,0} \quad \dots \quad F^0_{r_0-1,0}}_{\texttt{HydroUnitBlock } 0} \quad \dots \quad \underbrace{F^k_{0,0} \quad \dots \quad F^k_{r_k-1,0}}_{\texttt{HydroUnitBlock } k} \quad \dots \quad \underbrace{F^{H-1}_{0,0} \quad \dots \quad F^{H-1}_{r_{H-1}-1,0}}_{\texttt{HydroUnitBlock } H-1} \Big]^\top. \tag{7}$$

Since the mapping matrix A is very sparse (it is the identity matrix), it would be convenient to use the sparse representation of the mapping. The number of rows and columns of the mapping matrix are equal to the total number of reservoirs. Also the number of nonzero entries in the mapping matrix is equal to that number. Therefore, in the specification of the `BendersBFunction`, we would have

$$\texttt{NumVar} = R, \quad \texttt{NumRow} = R, \quad \text{and} \quad \texttt{NumNonzero} = R,$$

where R is the total number of reservoirs, i.e., $R = \sum_{i=0}^{H-1} r_i$. Since the A matrix is the identity matrix, the *variables* `NumNonzeroAtRow`, `Column`, and `A` do not need to be provided. But just as an exercise, we explain how the A matrix could be provided in terms of the *variables* `NumNonzeroAtRow`, `Column`, and `A`. Since there is a single nonzero entry at each row, the *variable* `NumNonzeroAtRow` would be a vector of size `NumRow` (i.e., R) with all entries equal to 1:

$$\texttt{NumNonzeroAtRow} = (1, \dots, 1).$$

The *variable* `Column` would be the vector

$$\texttt{Column} = (0, 1, \dots, R-1)$$

and the *variable* A, containing the nonzero entries of the mapping matrix, would be the vector of size NumNonzero (i.e., R) given by

$$\mathtt{A} = (1, \ldots, 1).$$

Finally, the *variable* b, containing the constant term of the vector, would be the vector of size NumRow (i.e., R) given by

$$\mathtt{b} = F,$$

with F given by (7).

Now, to each row of the mapping, we must associate the corresponding RowConstraint. In a HydroUnitBlock, constraints (6) belong to a two-dimensional group of RowConstraints which is indexed over the dimensions (inner) TimeHorizon and NumberReservoirs. Thus, in that group of RowConstraints, the j-th constraint in (6) is represented by the RowConstraint whose (two-dimensional) index is $(0, j)$. In the specification of the BendersBFunction, each of those RowConstraints would be indicated by an AbstractPath starting from the BendersBlock, i.e., having the BendersBlock as the reference Block.

Considering the structure of the BendersBFunction illustrated by Figure 2 and the structure of the UCBlock depicted in Figure 3, assuming the HydroSystemUnitBlock is the first UnitBlock in the UCBlock, an AbstractPath from the UCBlock (which is the inner Block of the BendersBFunction) to the "final volume" Constraint with (two-dimensional) index $(0, j)$ of the i-th HydroUnitBlock would be given by

$$B(0) \to B(i) \to C(0, j), \tag{8}$$

where $B(k)$ indicated the k-th nested Block of a Block and $C(k_1, k_2)$ indicates the Constraint whose element index is k_2 within the group of Constraints whose index is k_1.

To understand this path, we can start from the RowConstraint of interest and navigate up to the reference Block (which is the UCBlock). Since the two-dimensional index of the target RowConstraint is $(0, j)$, its element index is j according to the definition of "element index" of a Constraint in an AbstractPath (see (1) for details). Moreover, that RowConstraint belongs to the group of Constraints whose index is 0 (the "final volume" constraints are the first group of Constraints in a HydroUnitBlock; see the comments of the HydroUnitBlock class for details). Hence, we have the node $C(0, j)$. The target RowConstraint belongs to the i-th HydroUnitBlock, i.e., the i-th nested Block of the HydroSystemUnitBlock. Hence, we have the node $B(i)$. Finally, the HydroSystemUnitBlock is the first nested Block of the UCBlock. Hence, we have the node $B(0)$.

The AbstractPaths of those RowConstraints must be given as a vector of AbstractPath. From the general path (8) to a RowConstraint, it is straightforward to construct a vector of AbstractPaths to those RowConstraints. By concatenating the sequence of nodes of all AbstractPaths, we would have the following vectors of size $3R$:

$$\begin{aligned}
\mathtt{PathNodeTypes} &= (\text{'B'}, \text{'B'}, \text{'C'}, \ \ldots, \ \text{'B'}, \text{'B'}, \text{'C'}) \\
\mathtt{PathElementIndices} &= (\mathtt{inf}, \mathtt{inf}, 0, \ \ldots, \ \mathtt{inf}, \mathtt{inf}, r_0 - 1, \\
&\qquad \mathtt{inf}, \mathtt{inf}, 0, \ \ldots, \ \mathtt{inf}, \mathtt{inf}, r_1 - 1, \\
&\qquad \ldots, \\
&\qquad \mathtt{inf}, \mathtt{inf}, 0, \ \ldots, \ \mathtt{inf}, \mathtt{inf}, r_{H-1} - 1) \\
\mathtt{PathGroupIndices} &= (\underbrace{0, 0, 0, \ \ldots, \ 0, 0, 0,}_{r_0 \text{ times}} \ \underbrace{0, 1, 0, \ \ldots, \ 0, 1, 0,}_{r_1 \text{ times}} \ \ldots, \ \underbrace{0, H-1, 0, \ \ldots, \ 0, H-1, 0}_{r_{H-1} \text{ times}}).
\end{aligned}$$

Notice how PathGroupIndices is constructed. It is formed by $H - 1$ sub-vectors. The i-th sub-vector is associated with the i-th HydroUnitBlock, has size $3r_i$, and is formed by concatenating $(0, i, 0)$ r_i times. Since each AbstractPath has three nodes and there are a total of R AbstractPaths, we would have

$$\mathtt{PathDim} = R, \quad \mathtt{TotalLength} = 3R, \quad \text{and} \quad \mathtt{PathStart} = (0, 3, 6, \ldots, 3(R-1)).$$

Lastly, since all those RowConstraints represent equality constraints, the ConstraintSide *variable* would be a vector of size NumRow (i.e., R) with all elements equal to 'B', meaning that both sides of the RowConstraint are equally affected:

$$\mathtt{ConstraintSide} = (\text{'B'}, \ldots, \text{'B'}).$$

But since all elements are equal to 'B', the ConstraintSide *variable* does not need to be provided.

8.1.2 StochasticBlock

The *group* Block of the StochasticBlock must contain the description of the associated BendersBlock. Moreover, the StochasticBlock also requires a vector of SimpleDataMapping which, in the case of the SSV, defines how the data of the UCBlock is updated considering a given sub-scenario. A sub-scenario is a vector of double containing a realization of the stochastic data of the UCBlock. Each SimpleDataMapping (see §3.3.2) would be associated with a particular part of the sub-scenario and a particular stochastic data of some :Block. A SimpleDataMapping requires five elements:

- A :Block whose data is affected.
- The name of the function that will be responsible for updating the data of that :Block.
- An ordered multiset of indices that specifies which elements of the sub-scenario should be considered. This multiset must therefore contain elements between 0 and $n-1$, where n is the size of the sub-scenario. We call this multiset the SetFrom multiset.
- An ordered multiset of indices that specifies which elements of the :Block will be updated. We call this multiset the SetTo multiset.
- The type of the data that will be updated.

(A multiset is a set that can contain multiple instances of the same element. For instance, $\{2, 2, 4, 4, 7\}$ is a multiset with cardinality 5. An ordered multiset is a multiset in which the order of the elements matter. For instance, $\{2, 2, 4, 4, 7\}$ and $\{2, 4, 4, 2, 7\}$ are different multisets. In most cases, if not all, the SetFrom and SetTo multisets will be simple sets.) The SetFrom multiset defines a mapping $\mathcal{M} : \mathbb{R}^n \to \mathbb{R}^m$, where n is the size of the sub-scenario and m is the cardinality of the SetFrom multiset. Let s_i denote the i-th element of the SetFrom multiset. For a given sub-scenario ξ, $\mathcal{M}(\xi)$ is the vector whose i-th element is ξ_{s_i}. The SetTo multiset will be used to associate each element of $\mathcal{M}(\xi)$ to a particular element of in the data structure of the :Block.

Consider, for example, a UCBlock that has N UnitBlocks, one of them being a HydroSystemUnitBlock and two of them being ThermalUnitBlocks. Suppose that the maximum power of the first ThermalUnitBlock is stochastic for the inner stages 2, 3, and 5 and that the maximum power of the second ThermalUnitBlock is stochastic for the inner stages 0, 1, 2, and 3. Suppose also that the NetworkBlock associated with the inner stage 1 has 10 nodes, indexed from 0 to 9, and that the active demand at the node with index 4 is stochastic. Finally, suppose that the fourth HydroUnitBlock of the HydroSystemUnitBlock has some stochastic inflows. The inflow occurs at each reservoir of a HydroUnitBlock at each inner stage. To simplify, let us suppose that the inflows at the second reservoir at the inner stage 0 and the inflows at the fifth reservoir at the inner stage 15 are stochastic.

We could define a sub-scenario ξ as a vector in $\mathbb{R}^{10}$ such that

- (ξ_0, ξ_1, ξ_2) represent the maximum power of the first ThermalUnitBlock for the inner stages 2, 3, and 5.
- $(\xi_3, \xi_4, \xi_5, \xi_6)$ represent the maximum power of the second ThermalUnitBlock for the inner stages 0, 1, 2, and 3.
- (ξ_7) represent the active demand of the NetworkBlock associated with the inner stage 1 at node with index 4.
- (ξ_8, ξ_9) represent the inflows at the second and fifth reservoirs of the fourth HydroUnitBlock at the inner stages 0 and 15, respectively.

In this case, we can construct four SimpleDataMappings to update the stochastic data of these Blocks and one SimpleDataMapping to update the BendersBFunction (which depends on the inflow at the first inner stage). First of all, here is a list of functions that are registered in the methods factory. In the context of the SSV, they can be used to define the stochastic data (through the SimpleDataMapping) as we will see in the sequel.

- DCNetworkBlock::set_active_demand
  This function updates the active demand of a NetworkBlock. The active demand is represented by a one-dimensional array whose i-th element is the active demand at the i-th node.
- ThermalUnitBlock::set_maximum_power
  This function updates the maximum active power output of a ThermalUnitBlock. The maximum active power output is represented by a one-dimensional array whose t-th element is the maximum active power output at the inner stage t.

- `HydroUnitBlock::set_inflow`
  This function updates the inflow of a `HydroUnitBlock`. The inflow is represented by a two-dimensional array whose element at position (i, t) is the inflow at node i at the inner stage t.

- `IntermittentUnitBlock::set_maximum_power`
  This function updates the maximum potential power production of a `IntermittentUnitBlock`. The maximum potential power production is represented by a one-dimensional array whose t-th element is the maximum potential power production at the inner stage t.

- `BendersBFunction::modify_constants`
  This function updates the constant b of the mapping of the `BendersBFunction` (see (3) in §4.10).

Let us begin with the first `ThermalUnitBlock`. Its data is given in the first two entries of the sub-scenario and is associated with the maximum power at the inner stages 2, 3, and 5. Therefore, we have $\{0, 1, 2\}$ as the `SetFrom` set and $\{2, 3, 5\}$ as the `SetTo` set. Since the maximum power is of type `double`, its data type is 'D'. The name of the function responsible for updating the maximum power is `ThermalUnitBlock::set_maximum_power`. Finally, we need to specify that this data is associated with our first `ThermalUnitBlock`. This is done by providing an `AbstractPath` to that `Block`. Suppose that this `ThermalUnitBlock` is the second sub-`Block` of the `UCBlock` so that its group index is 1 (this `ThermalUnitBlock` would be the `UnitBlock` with index 1 in Figure 3 and associated with *group* `UnitBlock_0` in the specification of the `UCBlock`; see §4.2). The path from the `BendersBlock` to this `ThermalUnitBlock` would then be given by

$$O \to B(0) \to B(1).$$

Analogously, for the second `ThermalUnitBlock`, we would have $\{3, 4, 5, 6\}$ as the `SetFrom` set, $\{0, 1, 2, 3\}$ as the `SetTo` set and, assuming it is the fourth sub-`Block` of the `UCBlock` (and thus its group index is 5), the path to this `Block` would be

$$O \to B(0) \to B(5).$$

For the `NetworkBlock`, we would have $\{7\}$ as the `SetFrom` set and $\{4\}$ (the index of the node) as the `SetTo` set. The name of the function that updates this data is `DCNetworkBlock::set_active_demand` and the data is also of type `double` (indicated by 'D'). Finally, we need to specify the path from the `BendersBlock` to this `NetworkBlock`. As we see in Figure 3, the `NetworkBlocks` come after the `UnitBlocks`. Since in our example the `UCBlock` has N `UnitBlocks` and the `NetworkBlock` of interest is associated with stage 1 (the stages being indexed from 0 to T^{in}), the group index of this `NetworkBlock` would be $N+1$. Therefore, the path from the `BendersBlock` to this `NetworkBlock` would be given by

$$O \to B(0) \to B(N + 1).$$

For the `HydroUnitBlock`, we would have $\{8, 9\}$ as the `SetFrom`. Since the inflow of a `HydroUnitBlock` is associated with a two-dimensional data structure, the specification of the `SetTo` is a little bit more complicated (but not too much). This data structure – let us call it `inflow` – is such that `inflow[i][t]` gives the inflow at reservoir i at the inner stage t. As you may notice, a single inflow is identified by two indices (the index of the reservoir and the inner stage) while `SetTo` is a set of simple indices. So, we must convert a pair of indices into a single index. The conversion is made considering that a multi-dimensional structure is vectorized by making its last dimension vary faster. This means that for a two-dimensional matrix A with m rows $a_0, \ldots, a_{m-1}$ and n columns, its vectorized form is given by arranging its rows in sequence: $(a_0, a_1, \ldots, a_{m-1})$. Then, the element at position (i, j) in the matrix A is the element at position $i \times n + j$ in its vector representation. See (1) for the index of an element in an arbitrarily multi-dimensional array. The pairs of indices that we need to convert are $(1, 0)$ (the second reservoir, whose index is 1, and the inner stage 0) and $(4, 15)$ (the fifth reservoir, whose index is 4, and the inner stage 15). Since the fourth `HydroUnitBlock` has r_3 reservoirs and the inner time horizon is T^{in}, the data structure representing the inflows is a $r_3 \times T^{\text{in}}$ two-dimensional array. Thus, the single indices associated with $(1, 0)$ and $(4, 15)$ are T^{in} and $4T^{\text{in}} + 15$, respectively. Thus, we would have $\{T^{\text{in}}, 4T^{\text{in}} + 15\}$ as the `SetTo` set. The name of the function that updates the inflow is `HydroUnitBlock::set_inflow` and the data is of type `double` (indicated by 'D'). Finally, we need to provide the path from the `BendersBlock` to this `HydroUnitBlock`. We are assuming that the `HydroSystemUnitBlock` is the first `Block` in the `UCBlock`. Therefore, since our `HydroUnitBlock` of interest has index 3 (it is the fourth `HydroUnitBlock` of the `HydroSystemUnitBlock`), the path would be given by

$$O \to B(0) \to B(0) \to B(3).$$

SetFrom	SetTo
$\{0, 1, 2\}$	$\{2, 3, 5\}$
$\{3, 4, 5, 6\}$	$\{0, 1, 2, 3\}$
$\{7\}$	$\{4\}$
$\{8, 9\}$	$\{T^{\text{in}}, 4T^{\text{in}} + 15\}$
$\{8\}$	$\{r_0 + r_1 + r_2 + 1\}$

Table 7 The `SetFrom` and `SetTo` sets.

As we saw in §8.1.1, the mapping of the `BendersBFunction` depends on the inflows at the first inner stage. In our example, the inflow at the second reservoir of the fourth `HydroUnitBlock` at the inner stage 0 is stochastic. Therefore, we also need a `SimpleDataMapping` to update the mapping of the `BendersBFunction`. In that mapping, the inflows appear in the vector F defined in (7). In F, the initial inflow at the second reservoir of the fourth `HydroUnitBlock` is linked with index $r_0 + r_1 + r_2 + 1$. Thus, in the `SimpleDataMapping` used to update the `BendersBFunction` we would have $\{8\}$ as the `SetFrom` set (since the initial inflow at the second reservoir of the fourth `HydroUnitBlock` is given by ξ_8) and $\{r_0+r_1+r_2+1\}$ as the `SetTo` set. The name of the function that updates the constant part of the mapping of the `BendersBFunction` is `BendersBFunction::modify_constants` and the data is of type `double` (indicated by 'D'). The path from the `BendersBlock` to the `BendersBFunction` is given by

$$O.$$

In the specification of the `StochasticBlock`, these `SimpleDataMappings` must be provided as a vector of `SimpleDataMapping` (see §3.3.2.2). Since there are five `SimpleDataMappings`, we have `NumberDataMappings` = 5. All stochastic data have a type `double`, so `DataType` does not need to be provided. Anyway, it could be specified as

$$\texttt{DataType} = (\text{'D'}, \text{'D'}, \text{'D'}, \text{'D'}, \text{'D'}).$$

All multisets can be provided as `Subsets`. However, since the representation of a `Range` is more compact for sets with at least three elements, we will use it whenever is convenient. Table 7 shows the sets that we need to provide. The first two `SetFrom` sets and the second `SetTo` set have at least three elements, so we will use a `Range` to represent each of them. We begin with the `SetSize` *variable*. This one-dimensional *variable* indicates the size or the type of each set. For a `Subset`, we must include its size in `SetSize`. For a `Range`, we must include the number 0 in `SetSize`. Thus, in our example, this *variable* would be given as

$$\texttt{SetSize} = (0, 3, \quad 0, 0, \quad 1, 1, \quad 2, 2, \quad 1, 1).$$

Recall that the elements at positions $2i$ and $2i + 1$ correspond to the sizes (or types) of the i-th `SetFrom` and `SetTo` sets. Now, we must specify the elements that belong to each set. For a `Range` representing the set $[a, b)$, we must provide a and b. For a `Subset`, we must provide its elements. Thus, the `SetElements` *variable* would be given as

$$\texttt{SetElements} = (0, 3, 2, 3, 5, \quad 3, 7, 0, 4, \quad 7, 4, \quad 8, 9, T^{\text{in}}, 4T^{\text{in}} + 15, \quad 8, r_0 + r_1 + r_2 + 1).$$

(The first `SimpleDataMapping` has $\{0, 1, 2\}$ as its `SetFrom` set and $\{2, 3, 5\}$ as its `SetTo` set. In this case, the `SetFrom` can be represented as the `Range` $[0, 3)$. The `SetTo` cannot be represented as a `Range` (it is not an interval of integers), so we need to explicitly list all its elements. Thus, the first elements of `SetSize` are 0 and 3 (indicating that the first `SetFrom` set is a `Range` and the first `SetTo` set is a `Subset` with three elements). Moreover, the first elements of `SetElements` are 0 and 3 (describing the first `SetFrom`) and 2, 3, and 5 (describing the first `SetTo`). The next elements of `SetSize` and `SetElements` are those describing the `SetFrom` and `SetTo` sets of the second `SimpleDataMapping`. In the second `SimpleDataMapping`, we have $\{3, 4, 5, 6\}$ as the `SetFrom` set and $\{0, 1, 2, 3\}$ as the `SetTo` set. These sets can be represented as the `Ranges` $[3, 7)$ and $[0, 4)$, respectively. So, the next elements of `SetSize` are 0, 0 (indicating that both sets are `Ranges`) and the next elements of `SetElements` are 3, 7, 0, 4. And so on.)

The `FunctionName` *variable* must contain the names of the functions (registered in the "methods factory")

responsible for updating the data of the Blocks. In our example, we would have

$$\begin{aligned}\texttt{FunctionName} = (&\text{“}\texttt{ThermalUnitBlock::set\_maximum\_power}\text{”},\\ &\text{“}\texttt{ThermalUnitBlock::set\_maximum\_power}\text{”},\\ &\text{“}\texttt{DCNetworkBlock::set\_active\_demand}\text{”},\\ &\text{“}\texttt{HydroUnitBlock::set\_inflow}\text{”},\\ &\text{“}\texttt{BendersBFunction::modify\_constants}\text{”}).\end{aligned}$$

We must also provide a vector of AbstractPath to the affected Blocks and the BendersBFunction. Since we have five SimpleDataMappings, we also have five paths. So, we have PathDim = 5. In our example, the first three paths have size 3, the fourth path has size 4, and the last path has size 1. So, we have

$$\texttt{TotalLength} = 14 \quad \text{and} \quad \texttt{PathStart} = (0, 3, 6, 9, 13).$$

We also have

$$\begin{aligned}\texttt{PathNodeTypes} &= (\text{‘O’}, \text{‘B’}, \text{‘B’}, \quad \text{‘O’}, \text{‘B’}, \text{‘B’}, \quad \text{‘O’}, \text{‘B’}, \text{‘B’}, \quad \text{‘O’}, \text{‘B’}, \text{‘B’}, \text{‘B’}, \quad \text{‘O’})\\ \texttt{PathGroupIndices} &= (\texttt{inf}, 0, 1, \quad \texttt{inf}, 0, 5, \quad \texttt{inf}, 0, N+1, \quad \texttt{inf}, 0, 1, 3, \quad \texttt{inf})\\ \texttt{PathElementIndices} &= (\texttt{inf}, \texttt{inf}, \texttt{inf}, \quad \texttt{inf}, \texttt{inf}, \texttt{inf}, \quad \texttt{inf}, \texttt{inf}, \texttt{inf}, \quad \texttt{inf}, \texttt{inf}, \texttt{inf}, \texttt{inf}, \quad \texttt{inf}).\end{aligned}$$

Since all elements of PathElementIndices are equal to inf, this *variable* does not need to be provided. Finally, we need to inform the type of each affected object. The first four objects are Blocks while the last one is a Function. Thus, we have

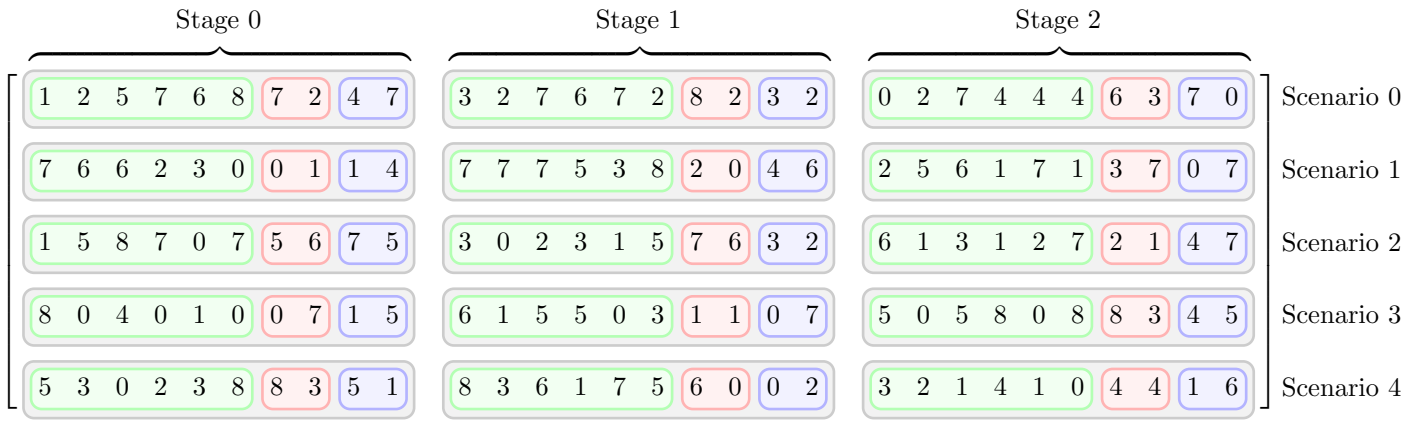
$$\texttt{Caller} = (\text{‘B’}, \text{‘B’}, \text{‘B’}, \text{‘B’}, \text{‘F’}).$$

8.1.3 SDDPBlock

We now describe the remaining part of the SDDPBlock. In the beginning of §8.1, we have seen how the scenarios are specified in the SDDPBlock. According to our notation, the *dimension* TimeHorizon of SDDPBlock would be equal to T^{out}. Following the example of §8.1.2, each sub-scenario (a part of the scenario that is associated with a particular outer stage) has size 10. Assuming all sub-scenarios have the same size, there would be no need to provide the *dimension* SubScenarioSize and the *dimension* ScenarioSize would be equal to $10T^{\text{out}}$ (for each stage $t \in \{0, \ldots, T^{\text{out}} - 1\}$, the size of a sub-scenario associated with stage t would be 10). Also, assuming there is a total of m scenarios, the *dimension* NumberScenarios would be equal to m. In our example, we can identify three groups of related random data: maximum power of a ThermalUnitBlock, active demand of a NetworkBlock, and inflows of a HydroUnitBlock. Thus, the *dimension* NumberRandomDataGroups would be equal to 3. The first group would then have size 6 (the maximum power of both affected ThermalUnitBlocks) and the second and third groups would have each one size 2. Therefore, the SizeRandomDataGroups *variable* would be given by

$$\texttt{SizeRandomDataGroups} = (6, 2, 2).$$

For instance, if we had $T^{\text{out}} = 3$ and $m = 5$, the Scenarios *variable* could be the following matrix:



The green, red, and blue boxes indicate the maximum power of the `ThermalUnitBlocks`, the active demand of the `NetworkBlock`, and the inflows of the `HydroUnitBlock`, respectively. The gray box indicates a sub-scenario.

The last sub-`Block` of a `HydroSystemUnitBlock` is a `PolyhedralFunctionBlock` (see §4.5) which contains a `PolyhedralFunction` (see §4.5.1). For each $t \in \{0, \dots, T^{\text{out}}-2\}$, the `PolyhedralFunction` associated with the outer stage t represents an approximation to the (unknown) expected future cost function. This is the function that is updated at each step of the SDDP method. This function depends on the volumes of all reservoirs at the last inner stage ($T^{\text{in}}-1$). Following the same notation used to describe constraints (6), this function depends on the variables $v^i_{j,T^{\text{in}}-1}$ for each `HydroUnitBlock` $i \in \{0, \dots, H-1\}$ and for each reservoir $j \in \mathcal{N}^i$ of that `HydroUnitBlock`. The only thing that must be provided in the specification of the `PolyhedralFunction` is the *dimension* `PolyFunction_NumVar`. This *dimension* must be equal to the total number of reservoirs (which is R in our example).

If one wishes to provide an initial approximation of the future cost function, then other *variables* and *dimensions* must also be provided. The `PolyhedralFunction` represents a function f of the form

$$f(x) = \max_{k \in \{0,\dots,p-1\}} a_k^\top x + b_k \tag{9}$$

or

$$f(x) = \min_{k \in \{0,\dots,p-1\}} a_k^\top x + b_k. \tag{10}$$

The *dimension* `PolyFunction_sign` indicates whether the `PolyhedralFunction` has the form (9), and therefore is a convex, or (10), and thus is concave. In the SSV, the future cost function is convex and has the form (9), so that `PolyFunction_sign` should be `true`. The *dimension* `PolyFunction_NumRow` indicates the number of affine functions that define the polyhedral function. In (9) and (10), this number is p. The vectors a_k and the constants b_k are provided in the `PolyFunction_A` and `PolyFunction_b` *variables*, respectively. `PolyFunction_A` must contain a matrix $A \in \mathbb{R}^{p \times R}$ such that, for each $k \in \{0, \dots, p-1\}$, a_k is its k-th row, and `PolyFunction_b` is a one-dimensional array whose k-th element is b_k. Finally, it is also possible to indicate a global lower (or upper) bound on the value of the function by setting the `PolyFunction_lb` *variable*. In the SSV, a valid global lower bound on the value of the future cost function is zero.

In the specification of the `SDDPBlock`, it is necessary to specify a path to each of these `PolyhedralFunctions`. For each outer stage $t \in \{0, \dots, T^{\text{out}}-2\}$, the required path is an `AbstracPath` from the `BendersBlock` associated with stage t to the `PolyhedralFunction`. Recall that we assumed that the `HydroSystemUnitBlock` is the first sub-`Block` of the `UCBlock`, so that its group index is 0. Thus, the path from the `BendersBlock` to the `PolyhedralFunction` would be the following:

$$O \to B(0) \to B(0) \to B(H). \tag{11}$$

We have seen in §8.1.2 an example on how to specify a vector of `AbstractPaths`. Here we assume that all paths are equal (that is, we assume that the `HydroSystemUnitBlock` is the first sub-`Block` of `UCBlock` and that the `PolyhedralFunctionBlock` is the last sub-`Block` of the `HydroSystemUnitBlock`). In this case, a vector with a single `AbstractPath` can be provided. Thus, the `PathDim` *dimension* would be 1 and, according (11), we would have `TotalLength` $= 4$, `PathStart` $= (0)$ and

$$\begin{aligned} \texttt{PathNodeTypes} &= (\text{'O'}, \text{'B'}, \text{'B'}, \text{'B'}) \\ \texttt{PathGroupIndices} &= (\texttt{inf}, 0, 0, H) \\ \texttt{PathElementIndices} &= (\texttt{inf}, \texttt{inf}, \texttt{inf}, \texttt{inf}). \end{aligned}$$

Again, since all elements of `PathElementIndices` are equal to `inf`, this *variable* does not need to be provided.

The `SDDPBlock` also requires an admissible state for each outer stage. In the SSV, a state for a given stage is formed by the initial volumes of all the reservoirs at that stage. Thus, the `StateSize` would be equal to R, the total number of reservoirs. An admissible state for stage $t \in \{0, \dots, T^{\text{out}}-1\}$ is a state for which the sub-problem at stage t is feasible. Trivial examples of non-admissible states include those in which the volumes of the reservoirs are too low (so that the demand may not be satisfied) or too high (so that the maximum volume limit of the reservoir may be exceeded if there is enough inflow). The state is a vector that must follow the same convention as that chosen for the vector x defined in (5). So, a state s^t for the outer stage $t \in \{0, \dots, T^{\text{out}}-1\}$ must be a vector in $\mathbb{R}^R$ with the form

$$s^t = \Big[\underbrace{\begin{matrix} s^t_{0,0} & \cdots & s^t_{0,r_0} \end{matrix}}_{\texttt{HydroUnitBlock } 0} \cdots \underbrace{\begin{matrix} s^t_{i,0} & \cdots & s^t_{i,r_i} \end{matrix}}_{\texttt{HydroUnitBlock } i} \cdots \underbrace{\begin{matrix} s^t_{H-1,0} & \cdots & s^t_{H-1,r_{H-1}} \end{matrix}}_{\texttt{HydroUnitBlock } H-1} \Big] \tag{12}$$

where $s^t_{i,j}$ is the initial volume for the j-th reservoir of the i-th `HydroUnitBlock`. The `AdmissibleState` *variable* would then be a one-dimensional array of size RT^{out} containing the states for all outer stages arranged as follows:

$$\texttt{AdmissibleState} = (s^0, \ldots, s^{T^{\text{out}}-1}),$$

where $s^t \in \mathbb{R}^R$ is given by (12) for each $t \in \{0, \ldots, T^{\text{out}} - 1\}$.

9 Conclusion

This document is intended as a quick reference manual for anybody who is required to construct SMS++ NETCDF input files. It should be remarked that basically all the `:Block` and `:Configuration` objects are supposed to have an in-memory interface to initialize the data. Thus, a workable way to produce NETCDF input files for those is to use the in-memory interface, and then the `serialize()` method to save it into a file. Yet, NETCDF has interfaces for all primary programming languages, hence files with the required format can be written independently by any SMS++ object.

References

[1] Unidata "The NETCDF software", http://doi.org/10.5065/D6H70CW6, 2019.