NANDRAD Model Reference

Andreas Nicolai
andreas.nicolai@tu-dresden.de

Version 1.0.0, July 2020

Table of Contents

This document contains a description of the various implemented models and the parametrization in the NANDRAD project file. It is therefore both input reference and model description.

1. Overview

1.1. Simulation time and absolute time reference

- how is the simulation time defined
- how is the absolute time obtained (start year and start time)
- how are cyclic annual simulations handled, how are continuous multi-year simulations handled

2. Models and Input Reference

2.1. Climatic loads

2.1.1. Overview

The climatic loads model is a purely time-dependent model without other input dependencies. For solar radiation calculation, it needs information on the building location. Also, for most simulations, a climate data file is needed.

2.1.2. Specification

Information about location and climate data is stored in the `Location` section of the project file:

```
<Location>
    <IBK:Parameter name="Latitude" unit="Deg">51</IBK:Parameter>
    <IBK:Parameter name="Longitude" unit="Deg">13</IBK:Parameter>
    <IBK:Parameter name="Albedo" unit="-">0.2</IBK:Parameter>
    <IBK:Parameter name="Altitude" unit="m">100</IBK:Parameter>
    <ClimateFileName>${Project Directory}/climate/testClimate.epw</ClimateFileName>
</Location>
```

The parameter `Albedo` is used for diffuse solar radiation calculation. `Altitude` is needed for specific altitude-related parameters (**TODO**).

Parameters `Latitude` and `Longitude` are optional if a climate data file is given, otherwise they override the location parameters stored therein (see Building/Station location).

Climate File Support

Currently, `c6b`, `wac` and `epw` files are supported (see CCM-Editor help).

Building/Station location

Typically, climate data files contain information on latitude and longitude of the weather station.

- If these parameters are not specified explicitely in the project file, the latitude/longitude from the climate data

file are used.

- If parameters are specified in the project file, always **both** parameters must be given (and be valid) and then the these parameters from the project file are used instead of the climate data file location parameters.

Additional radiation sensors

It is possible to specify additional planes (sensors) to generate solar radiation load outputs.

TODO

2.1.3. Implementation

The `Loads` model is a pre-defined model that is always evaluated first whenever the time point has changed. It does not have any other dependencies.

It provides all resulting variables as `constant` (during iteration) result variables, which can be retrieved and utilized by any other model.

With respect to solar radiation calculation, during initialization it registers all surfaces (with different orientation/inclination) and provides an ID for each surface. Then, models can request direct and diffuse radiation data, as well as incidence angle for each of the registered surfaces.

Registering surfaces

Each construction surface (interface) with outside radiation loads registers itself with the Loads object, hereby passing the interface object ID as argument and orientation/inclination of the surface. The loads object itself registers this surface with the climate calculation module (CCM) and retrieves a surface ID. This surface ID may be the same for many interface IDs.

The Loads object stores a mapping of all interface IDs to the respective surface IDs in the CCM. When requesting the result variable's memory location, this mapping is used to deliver the correct input variable reference/memory location to the interface-specific solar radiation calculation object.

2.2. Schedules

2.2.1. Overview

Schedules provide purely time-dependent quantities, similar to climatic loads.

Different to other results-producing models, schedules generate variables for sets of dependent models. As such, a schedule is formulated for an object list, which selects a set of objects taking the provided values.

For example, a schedule defines heating set points (HeatingSetPoint) for living room zones. These are selected by an object list "Living room", which selects *Zone*-type objects with a certain ID range.

2.2.2. Defining schedules

Rules

A certain variable must be only defined once per object list

For example, if you have a regular daily-cycle-based schedule for "HeatingSetPoint" and zone object list "office spaces", there must not be an annual schedule for "HeatingSetPoint" and the same object list name "office spaces"

A variable must be defined unambiguously with respect to addressed object

For example, you may have a "HeatingSetPoint" for zone object list "office spaces" and this object list addresses zones with IDs 1 and 4. Now there is a second object list "all spaces", with wildcard ID=* (hereby addressing all zones). You **most not** define the variable "HeatingSetPoint" again for this object list, since otherwise you would get ambiguous defintions of this variable for zones 1 and 4.

Annual schedules must begin at simulation start (past the end, values are constant extrapolated)

For annual cyclic schedules, the schedule must start with time 0. For non-cyclic schedules, the schedule must start at latest at actual simulation start, so that start year ⇐ simulation start year and if same year, start time < simulation start time. Basically, the solver must be able to query a value at simulation start.

If simulation continues past the end of an annual schedule, the last value will be simply kept (constant extrapolation).

Regular schedules (based on daily cycles)

...

Annual schedules (as linearly interpolated splines)

Annual schedules are basically data tables with

2.2.3. Implementation

Schedules do not have any dependencies, and are not part of the model graph. They are updated just as climatic loads whenever time changes.

Instead of generated a (potentially large) set of variables for each object adressed by the object list, schedules provide result variable slots for each object list and scheduled quantity. The individual model instances requesting their scheduled parameters share the same variable slot.

For example, two zones of the same object list request a variable reference (pointer to variable slot) from the schedule object, and will get the same pointer for the same variable.

Schedules do not implement the regular model interfaces and are not included in the model graph. Instead, they are handled in a special way by the framework.

Variable lookup

1. Schedules define variables for object lists.
2. Object lists address a range of objects based on filter criteria, such as object reference type (e.g. Zone, ConstructionInstance, Interface), and id group/range (a set of IDs)

When a certain object (e.g. a zone with a given ID) wants to get access to a parameter defined for it, a `ValueReference` can be created with:

- reference type = `ZONE`
- id = zone-id
- variable_name = required scheduled parameter name

and the schedule object may then lookup the variable as follows:

- cycle through all known object lists (i.e. object lists used in schedule definitions)
- check if reference type matches, and if id-name is in ID group of object list
- if object list was found, resolve variable name (from enumeration `Results`)
- search map for this parameter name for a key that matches the object list's name
- if match was found, return offset/pointer to the respective result variable
- in all other cases, return nullptr

Variable lookup for outputs

When model input variables are resolved, schedules are the

3. Outputs: definitions, rules and implementation

3.1. Output definitions

Outputs are defined via:

- Grid (defines when outputs are to be written)
- Variable/Quantity ID name
- Object List (selects object to retrieve data from)
- Time handling (how to handle time averaging/integration)
- (optional) target filename

Example:

```
<!-- Definition of output grids -->
<Grids>
    <OutputGrid name="hourly">
        <Intervals>
            <Interval>
                <IBK:Parameter name="StepSize" unit="h">1</IBK:Parameter>
            </Interval>
        </Intervals>
    </OutputGrid>
</Grids>
<!-- Definition of outputs -->
<OutputDefinitions>
    <OutputDefinition>
        <Quantity>Temperature</Quantity>
        <ObjectListName>All zones</ObjectListName>
        <GridName>hourly</GridName>
    </OutputDefinition>
    <OutputDefinition>
        <Quantity>NaturalVentilationFlux</Quantity>
        <ObjectListName>All zones</ObjectListName>
        <GridName>hourly</GridName>
        <TimeType>Integral</TimeType>
    </OutputDefinition>
</OutputDefinitions>


<!-- Referenced object list -->
<ObjectLists>
    <ObjectList name="All zones">
        <FilterID>*</FilterID>
        <ReferenceType>Zone</ReferenceType>
    </ObjectList>
</ObjectLists>
```

`TimeType` is optional, same as `None` by default. `FileName` is optional (see below).

3.1.1. Output Grids

Output grids define *when* outputs are written. An output grid contains a list of intervals, with an output step size defined for each interval.

The intervals are expected to follow in consecutive order, optionally with a gap in-between.

Intervals can have up to 3 parameters:

- `Start` - the start time of the interval (see explanation below)
- `End`- the end time of the interval (see explanation below)
- `StepSize` - the distance between outputs within the interval

Time definitions

Time points in `Start` and `End` parameters are defined with respect to Midnight January 1st of the year in which the simulation starts.

Rules

- the `Start` parameter is optional, under the following conditions:
  - in the first interval, a missing `Start` parameter is automatically set to 0 (start of the year)
  - in all other intervals, the `End` time of the preceeding interval is taken (see next rule below)
- the end time of an interval is defined, either:
  - by defining the `End` parameter,
  - through definition of the `Start` parameter in next interval
  - through simulation end time in last das Intervall das letzte Intervall ist (in diesem Fall läuft das Interval bis zum Simulationsende) das darauffolgende Interval hat ein Start-Wert hat. Bei Angabe eines Start-Parameters muß der Zeitpunkt mindestens dem End-Zeitpunkt des vorangegangenen Intervalls entsprechen → die Intervalle sind sequentiell geordnet und überschneiden nicht. Die Angabe des Parameters StepSize ist zwingend.

3.1.2. Examples

- Requesting fluxes across construction interfaces: object list must reference interfaces
- Requesting energy supplied to layer in construction instance (floor heating): object list must reference construction instance, variable name must reference heat source + index of heat source (if several in construction): `<Quantity>HeatSource[1]</Quantity>` (first heat source in layer, counting from side A in construction, see Heat sources in constructions/layers).

3.2. Variable lookup rules

Quantities in output definitions define the ID names of the output quantities, optionally including an index notation when a single element of a vectorial quantity is requested. Hereby the following notations are allowed:

- `HeatSource[1]` - index argument is interpreted as defined by the providing models, so when the model provides a vector-valued quantity with model ID indexing, then the argument is interpreted as object ID (otherwise as positional index)
- `HeatSource[index=1]` - index argument is explicitely interpreted as position index (will raise an error when model provides quantity with model ID indexing)
- `HeatSource[id=1]` - index argument is explicitely interpreted as object ID (will raise an error when model provides quantity with positional indexing)

3.3. Output file generation rules

3.3.1. When no filename is given

Target file name(s) are automatically defined.

All outputs a grouped by:

- grid and quantity type (Load, Schedule, State, Flux, Misc), quantity type is predefined for most quantity IDs (unknowns → Misc)
- for each used grid:

- states → `states_<gridname>.tsv`
- loads → `loads_<gridname>.tsv`
- fluxes → `fluxes_<gridname>.tsv`
- fluxes (integrated) → `flux_integrals_<gridname>.tsv`

Special rule: when only one grid is used, and grid has hourly steps all year round, the suffix `_<gridname>` is omitted.

3.3.2. When a filename is given

- check: all output definitions using this filename must use the **same** grid (same time points for all columns required!)
- all quantities (regardless of type) are written to this file

3.4. Binary Format

First record: unsigned int - n (number of columns) Next n records: binary strings, leading size (unsigned int) and termination character (sanity checking)

Next ?? records: unsigned int - n (for checking) and afterwards n doubles

4. Implementation Concepts

4.1. Model objects, published variables and variable look-up

Physical models are implemented in model objects. That means the code lines/equations compute results based on input values.

Example 1. Model object

> A heating model takes room air temperature (state dependency) and a scheduled setpoint (time dependency) and computes a heating load as result.

The result is stored in a **persistent memory location** where dependent models can directly access this value.

4.1.1. Model instances

There can be several model instances - the actual object code resides only once in solver memory, but the functions are executed several times for each individual object (=model instance).

Example 2. Model instance

> You may have a model that computes heat flux between walls and zones. For each wall-zone interface, an object is instantiated.

Each model instance stores its result values in own memory.

4.1.2. Model results

Publishing model results

The model instances/objects must tell the solver framework what kind of outputs they generate. Objects generating results must derive from class `AbstractModel` (or derived helper classes) and implement the abstract interface functions.

`Schedules` and `FMIInputOutput` models are handled differently, when it comes to retrieving generated results.

The framework first requests a reference type (prefix) via `AbstractModel::referenceType()` of the model object. This reference type is used to group model objects into meaningful object groups.

Example 3. Reference types

Typical examples are `MRT_ZONE` for zonal quantities, or `MRT_INTERFACE` for quantities (fluxes) across wall-surfaces.

Each invididual result variable is published by the model in an object of type `QuantityDescription`. The framework requests these via call to `AbstractModel::resultDescriptions()`. Within the `QuantityDescription` structure, the model stores for each computed quantity the following information:

- id-name (e.g. "Temperature")
- physical display unit (e.g. "C") - interpreted as "display unit", calculations are always done in base SI units
- a descriptive text (e.g. "Room air temperature") (optional, for error/information messages)
- physical limits (min/max values) (optional, may be used in iterative algorithms)
- flag indicating whether this value will be constant over the lifetime of the solver/integration interval

A note on units

The results are stored *always* in base SI units according to the IBK-UnitList. A well-behaving model will always store the result value in the basic SI unit, that is "K" for temperatures, "W" for thermal loads and so on (see `IBK::UnitList`).

The unit is really only provided for error message outputs and for checking of the base SI unit matches the base SI unit of the requested unit (as an additional sanity check).

Vector-valued results

Sometimes, a model may compute a vector of values.

Example 4. Vector results

A ventilation model may compute ventilation heat loads for a number of zones (zones that are identified via object lists).

When a so-called vector-valued quantity is generated, the following additional information is provided:

- whether index-based or id-based access is anticipated
- a list of ids/indexes matching the individual positions in the vector (the size of the vector is also the size of the vector-valued quantities)

Example 5. ID-access example

The aforementioned ventilation model may be assigned to zones with IDs 1,4,5,10 and 11. Then the resulting ventilation heat losses will be defined for those zones only. Hence, the published quantity will look like:

```
- name = "NaturalVentilationHeadLoad"
- unit = "W"
- description = "Natural ventilation heat load"
- index-key-type = "ID"
- ID-list = {1,4,5,10,11}
```

Strong uniqueness requirement

The id-names of quantities are unique within a model object, and vector-valued quantities may not have the same ID name as scalar quantities. More precisely, variables need to be globally unique (see lookup rules below). This means that when there are two models with the same model-reference-type, the variable names must be unique among all models with the same reference-type.

For example: from a zone parametrization several models are instantiated with the same reference-type `MRT_ZONE`, for example `RoomBalanceModel` and `RoomStatesModel`. The resulting variables in both models *must not* have the same ID name, in order to be uniquely identified on global scope.

Initializing memory for result values

Each model object is requested by the framework in a call to `AbstractModel::initResults()` to reserve memory for its result variables. For scalar variables this is usually just resizing of a vector of doubles to the required number of result values. For vector-valued quantities the actual size may only be known later, so frequently here just the vectors are created and their actual size is determined later.

Since the information collected in `initResults()` is also needed when publishing the results to the framework, the function `AbstractModel::initResults()` is called before `AbstractModel::resultDescriptions()`.

Convenience implementation

Scalar variables are stored in double variables of the model. When using the convenience implementation in `DefaultModel` these are stored in vector `m_results`.

Vector-valued variables are stored in consecutive memory arrays with size matching the size of the vector. When using the `DefaultModel` implementation, these are stored in `m_vectorValuedResults`, which is a vector of `VectorValuedResults`.

The `DefaultModel` implementation makes use of enumerations `Results` and `VectorValuedResults`.

4.1.3. Model inputs

Similarly as output variables, model objects need input variables. Models requiring input must implement the interface of class `AbstractStateDependency`.

Input variable requirements are published similarly to results when the framework requests them in a call to `AbstractStateDependency::inputReferences()`. The information on requested results is stored in objects of type `InputReference`. The data in class `InputReference` is somewhat similar to that of `QuantityDescription` but contains additional data regarding the expectations of the model in the input variable.

A model may request scalar variable inputs only, even if the providing model generates these as a vector-valued quantity. That means, a model has the choice to request access to the entire vector-valued variable (and will usually get the address to the start of the vector-memory space), or a single component of the vector. In the latter case, the index/model-ID must be defined in the `InputReference` data structure.

Example 6. Input reference to a zone air temperature inside a ventilation model

An input required by the ventilation model can be formulated with the follwing data:

```
- reference type = MRT_ZONE
- object_id = 15 (id of the zone)
- name = "AirTemperature"
```

Given that information, the framework can effectively look-up the required variables.

Once the variable has been found, the framework will tell the object the memory location by calling `AbstractStateDependency::setInputValueRef()`.

FMI Export (output) variables

When FMI export is defined, i.e. output variables are declarted in the FMI interface, a list of global variable IDs to be exported is defined. For each of these variables an input reference is generated (inside the `FMIInputOutput` model), just as for any other model as well.

Example 7. FMI Output Variable example

Suppose an FMI exports the air temperature from zone id=15 and for that purpose needs to retrieve the currently computed temperature from the zone state model. The FMI output variable would be named *Zone[15].AirTemperature* and the input reference would be created as in the example above. This way, the framework can simply provide a pointer to this memory slot to the `FMIInputOutput` model just as for any other model.

Outputs

When initializing outputs, any published variable can be collected. Outputs declare their input variables just as any other model object.

4.1.4. Variable lookup

The frameworks job is to collect all

Resolving persistant pointers to result locations

Later, when the framework connects model inputs with results, the framework requests models to provide persistant memory locations for previously published results. This is done by calling `AbstractModel::resultValueRef()`, which get's a copy of the previously exported `InputReference`.

In order to uniquely identify a result variable **within** a model, normally only two things are needed:

- the ID name of the variable,
- and, *only in the case of vector-valued quantities*, the index/id.

However, in some cases, a model may request a variable with the *same quantity* name, yet from two different objects (for example, the air temperature of neighbouring zones). In this case, the quantity name alone is not sufficient. Hence, the full input reference including object ID is passed as identifier (**A change from NANDRAD 1!**).

To identify an element within a vector-valued result it is not necessary to specify whether it should be index or id based - the model publishing the result defines whether it will be index or id based access.

Naturally, for scalar result variables the index/id property is ignored.

The `QuantityName` struct contains this information (a string and an integer value).

Now the model searches through its own results and tries to find a matching variable. In case of vector-valued quantities it also checks if the requested id is actually present in the model, and in case of index-based access, a check is done if the index is in the allowed range (0...size-1).

If a quantity could be found, the corresponding memory address is returned, otherwise a nullptr. The framework now can take the address and pass it to any object that requires this input.

Global lookup rules/global variable referencing

To uniquely reference a resulting variable (and its persistent memory location), first the actual model object/instance need to be selected with the following properties:

- the type of object to search for (= reference-type property), for example `MRT_ZONE` or `MRT_CONSTRUCTIONINSTANCE`
- ID of the referenced object, i.e. zone id oder construction id.

Some model objects exist only once, for example schedules or climatic loads. Here, the reference-type is already enough to uniquely select the object.

Usually, the information above does select several objects that have been created from the parametrization related to that ID. For example, the zone parameter block for some zone ID generates several zone-related model instances, all of which have the same ID. Since their result variables are all different, the framework simply searches through all those objects until the correct variable is found. These model implementations can be thought of as one model whose equations are split up into several implementation units.

The actual variable *within* the selected object is found by ID name and optionally vector-element id/index, as

described above.

The data is collected in the class `InputReference`:

- `ObjectReference` (holds reference-type and referenced-object ID)
- `QuantityName` (holds variable name and in case of vector-valued quantities also ID/index)

Also, it is possible to specify a `constant` flag to indicate, that during iterations over cycles the variable is to be treated constant.

If several model objects are addressed by the same reference-type and ID (see example with models from zone parameter block), the variable names must be unique among all of these models.

FMI Input variable overrides

Any input variable requested by any other model can be overridden by FMU import variables. When the framework looks up global model references, and an FMU import model is present/parametrized, then first the FMI generated quantity descriptions are checked. The FMU import variables are exported as global variable references (with ObjectReference). Since these are then the same global variable identifiers as published by the models, they are found first in the search and dependent models will simply store points to the FMU variable memory.

Examples for referenced input quantities

Setpoint from schedules

Schedules are defined for object lists. Suppose you have an object list name "Living rooms" and corresponding heating/cooling setpoints.

Now a heating model may be defined that computes heating loads for a given zone. The heating model is implemented with a simple P-controller, that requires zone air temperature and zone heating setpoint.

Definition of the input reference for the zone air temperature is done as in the example above. The setpoint will be similarly referenced:

```
- reference type = MRT_ZONE
- object_id = 15 (id of the zone)
- name = "HeatingSetPoint"
```

4.2. Model Initialization Procedure

4.2.1. Pre-Solver-Setup steps

The following steps are done when initializing the model:

- parsing command-line and handling early command line options
- setting up directory structure and message handler/log file
- creating `NANDRAD::Project` instance
- setting default values via call to `NANDRAD::Project::initDefaults()`

- reading the project file (only syntactical checks are done, and for IBK::Parameter static arrays with default units in keyword list, a check for compatible units is made), may overwrite defaults; correct units should be expected in the data model after reading the project file (maybe add suitable annotation to code generator?)
- merge similarly behaving construction instances (reduce data structure size)

From now on the project data structure remains unmodified in memory until end of solver runtime and persistent pointers can be used to address parameter sets. This means no vector resizing is allowed, no more data members may be added/removed.

- setup fast-access connections between parameter data sets (zones to interfaces and vice versa) in Project data structure

Now the actual model initialization starts.

4.2.2. Model Setup