Documentation for INCABuilder model builders - documentation is in development

Magnus Dahler Norling

February 22, 2019

1 Introduction

This is documentation for model developers. Documentation for model users and framework developers will be provided in separate documents (eventually).

Note 1. It is recommended for any model developers to take a look at the tutorials and experiment with making changes to them before reading all of the documentation. The tutorials will also provide the information about how to compile the models into finished exes.

2 Basic concepts

2.1 The model

The model object contains all immutable information about the model that will not change with each particular usage of the model. The model object contains lists of various model entities and information about these that are provided by the model developer in the model registration routine. The different model entities are presented in Table 2.1

Index sets	Parameters, inputs and equation results can index over one or more index sets. For instance, you may want to evalute the same equations in multiple contexts, such as once per reach in a river, and in that case you want an index set for these reaches.
Parameter groups	Parameter groups are collections of parameters. A parameter group can vary with an index set, making each parameter have a separate value for each index in the index set. The parameter group can also be a child group of another parameter group. In that case, each parameter in the group has a separate value for each pair of indexes from the index set of its group and the parent group. This can be chained to make parameters vary with as many index sets as one wants.
Parameters	Parameters are values used to tune the equations of the modell. Parameters do not vary over time. Currently four different types of parameters are supported: double precision floating point, unsigned integer, boolean and time.
Inputs	Inputs are forcings on the model that vary over time. An example of an input is a time series of daily air temperature that has been measured in the field. Inputs can also vary with index sets. For instance, one can have a separate air temperature series for each subcatchment area in the model.
Equations	The model has a set of equations that are executed in a specific order for every timestep of the model (see later for how the run order is determined). Each equation can look up the value of various parameters, inputs, results of other equations (both from the current timestep and earlier) and produces a single output value for each time it is evaluated. There are several types of equations. The main two are discrete timestep equations and ordinary differential (ODE) equations.
Solvers	Each solver contains a list ODE equations that it will solve over one timestep whenever it is run. The solver must be registered with additional information about e.g. which integrator method it will use.

Table 1: The model entities

2.2 The dataset

The dataset contains information about the specific setup of a model. This includes all the indexes of each index set, as well as all the specific values of the parameters and input series. It will also contain the result series after the model is run. The typical way of setting up a dataset is by reading it in from a parameter file and an input file. The file formats for these are provided in separate documentation.

3 Model structure

3.1 Index sets and indexes

Index sets is one of the fundamental concepts in the INCABuilder framework. Examples of index sets can be ”Reaches”, or ”Landscape units”. Parameters, equation results and inputs can index over index sets, so that the same equation is evaluated for many indexes, and with different parameter values.

Example 1. The equation ”Snow fall” depends on the input timeseries ”Actual precipitation” and ”Air temperature”, and on the parameter ”Canopy interception”. Say that ”Canopy interception” indexes over the index set ”Landscape units”. In this case, ”Snow fall” will be evaluated once per index in the ”Landscape units” index set, using the corresponding parameter value of ”Canopy interception”. If one of the inputs also index over ”Reaches”, then ”Snow fall” will be evaluated once per reach-landscape unit pair, and so on.

Which index sets each model has is fixed, and determined by the model developer. However, which indexes each index set contains can usually be determined by the model user, for instance by configuring a parameter file. Some model do however require a fixed set of particular indexes for some of its index sets. For instance INCA-N only works with three soil boxes, while INCA-C works with four.

There are two types of index sets

i Basic index sets. These have no additional structure.

ii Branched index sets. These contain additional connectivity information, and are typically used to encode river structures. Each index in a branched index set has a list of incoming branches, which are other indexes in the same index set.

3.2 Parameter structure

Each parameter belongs to a parameter group, which again can belong to another parameter group and so on. Moreover, each parameter group can index over an index set. This means that each parameter can in practice be viewed as a multidimensional array of parameter values, where each dimension in the array is indexed by one of the model’s index sets. It is possible for a parameter to index over the same index set multiple times. For instance, the PERSiST hydrology model [1] has a ”Percolation matrix” parameter where both the row and column in the matrix is indexed by the soil box.

Example 2. A PERSiST ”Percolation matrix”, from a parameter file. (See separate documentation for detailed description of the parameter file format).

```
"Percolation matrix" :
0.2 0.8 0.0
1.0 0.5 0.5
0.0 0.0 1.0

0.1 0.9 0.0
1.0 0.6 0.4
0.0 0.0 1.0
```

Here the parameter indexes over ”Landscape units” (signifying what matrix you are looking at) and ”Soils” twice (signifying which row and column). The percolation matrix is used says something about what proportion of the water will flow between each soil box, but we will not go into detail about that here.

The parameter group and index set structure of the parameters are entirely fixed by the model (developer), and can not be configured in the parameter file (though one can usually configure which indexes each index set has).

3.3 Input structure

An input is a timeseries (usually of measured data or preprocessed based on measured data). Inputs are used as forcings in many models. For instance, hydrological models often use precipitation timeseries to determine how much water enters the system at each day. Inputs can either be global for the system, or like parameters can index over one or more index sets. For instance, one may want to have a separate precipitation timeseries for each subcatchment (reach). Unlike with parameters, which index set each input indexes over is determined in the input file. This can sometimes change the equation structure of the model. For instance, if precipitation and air temperature does not depend on which subcatchment one are in, the framework may decide for some models that equations like ”Precipitation falling as snow” should only be evaluated once globally instead of per subcatchment. This does however also depend on which parameters these equations reference. If ”Precipitation falling as snow” references a parameter that indexes over the ”Reaches” index set, that equation will still evaluate once per reach.

3.4 Equation batch structure

During the main run of the model, the model will for each timestep evaluate all equations (sometimes called state variables) in a given order. Equations can also be evaluated for multiple indexes. For instance the equations having to do with flow or volume of the reach are evaluated for each reach index. Equations are sorted so that an equation is always evaluated after other equations that it uses the result values of. To facilitate this, the model builds an equation batch structure.

Example 3. The equation batch structure for the framework implementation of PERSiST.

```
**** Result Structure ****
[Reaches][Landscape units]
       -----
       Snow melt
       Rainfall
       -----

[Reaches][Landscape units][Soils]
       -----
       Percolation input
       Saturation excess input
       Input
       Water depth 1
       Saturation excess
       Water depth 2
       Evapotranspiration X3
       Evapotranspiration X4
       Evapotranspiration
       Water depth 3
       Total runoff
       Drought runoff
       Percolation out
       Water depth 4
       Runoff
       Runoff to reach
       Water depth
       -----

[Reaches][Landscape units]
       -----
       Snow fall
       Snow as water equivalent
       (Cumulative) Total runoff to reach
       Diffuse flow output
       -----

[Reaches]
       -----
       (Cumulative) Total diffuse flow output
       Reach flow input
       ----- (SOLVER: Reach solver)
       Reach time constant
       (ODE) Reach flow
       (ODE) Reach volume
       -----
       Reach velocity
       Reach depth
       -----
```

The model has an ordered list of batch groups, where each batch group has an ordered list of batches. Each batch group also has an ordered list of index sets that it indexes over. Each batch can either be a discrete batch or a solver batch. The model will evaluate each batch group at a time. Then for each combination of indexes in the index sets it evaluates each batch in the batch group in order. In discrete batches, each equation is evaluated in order. In solver batches, the ODE system will be integrated possibly evaluating each equation many times. In Example

3, the first three batch groups contain one batch each, while the last batch group contains three batches.

Note 2. The only time a batch group can have more than one batch is if it contains both a solver batch and one or more discrete batches (or possibly another solver batch using a different integrator). Two discrete batches next to each other that index over the same index sets would have been merged.

Example 4. Pseudocode for how the model evaluates the batch structure in Example 3.

```
for every timestep:
        for every reach R:
                for every landscape unit LU:
                        evaluate "Snow melt" and "Rainfall" for this LU in this R.
        for every reach R:
                for every landscape unit LU:
                        for every soil box S:
                                evaluate "Percolation input" to "Water depth" for this S
                                     in this LU in this R
        for every reach R:
                for every landscape unit LU:
                        evaluate "Snow fall" to "Diffuse flow output" for this LU in
                            this R
        for every reach R:
                evaluate "Total diffuse flow output" and "Reach flow input"
                run a solver that integrates the ODE system containing the equations "
                    Reach time constant" to "Reach volume" (possibly evaluating them
                    many times)
                evaluate "Reach velocity" and "Reach depth"
```

At the end of a model run, one can extract a timeseries of each of these equations for any combination of indexes that they index over. For instance one can look at the timeseries for ”Snow melt” in {”Langtjern”, ”Forest”}, or the timeseries for ”Water depth” in {”Langtjern”, ”Peatland”, ”Organic soil layer”} (given that these were indexes provided for this particular dataset).

Unlike parameters, equations can never index over one index set more than once. We will go through how the index sets of an equation is determined and how it is placed in a batch structure later.

4 Model building

4.1 Setting up a project and creating a model object

We refer to the quick start guide on the front page of the github repository as well as the tutorials on how to quickly get a project started.

Usually you will have one header (.h) file with the procedures you use to build your model, and one source file (.cpp) with the main functio of the C++ program, where you do all application-level operations such as creating and destroying model and dataset objects or reading in parameter values from files, running the model and so on.

Example 5. Example of a main.cpp

```
#define INCA_PRINT_TIMING_INFO 1

#include "../inca.h" //NOTE: This path is relative to the location of this file
#include "mymodel.h"

int main()
{
	const char *InputFile = "mytestinputs.dat";
	const char *ParFile  = "mytestparameters.dat";

	inca_model *Model = BeginModelDefinition();

	AddTestModel(Model);

	ReadInputDependenciesFromFile(Model, InputFile);

	EndModelDefinition(Model);

	PrintResultStrucuture(Model);

	inca_data_set *DataSet = GenerateDataSet(Model);

	ReadParametersFromFile(DataSet, ParFile);
	ReadInputsFromFile(DataSet, InputFile);

	RunModel(DataSet);

	u64 Timesteps = GetTimesteps(DataSet);
	PrintResultSeries(DataSet, "AX", {}, Timesteps);
}
```

Example 6. mymodel.h

```
static void
AddTestModel(inca_model *Model)
{
	auto Days = RegisterUnit(Model, "days");
	auto Dimensionless = RegisterUnit(Model);

	auto System = RegisterParameterGroup(Model, "System");
	RegisterParameterUInt(Model, System, "Timesteps", Days, 100);
	RegisterParameterDate(Model, System, "Start␣date", "1980-1-1");

	auto X = RegisterInput(Model, "X");
	auto A = RegisterParameterDouble(Model, System, "A", Dimensionless, 1.0);

	auto AX = RegisterEquation(Model, "AX", Dimensionless);

	EQUATION(Model, AX,
		return PARAMETER(A) * INPUT(X);
	)
}
```

Example 7. mytestparameters.dat

```
parameters:
"Timesteps":
10

"Start date":
"2019-02-22"

"A":
5.0
```

Example 8. mytestinputs.dat

```
timesteps:
10

inputs:
"X":
1 2 3 4 5 6 7 8 9 10
```

Example 5 shows a main function that you can usually keep unchanged throughout model development. The only thing you may want to change in it is the debug printout (`PrintResultSeries`) at the end, to print out different debug result series. Your main model development will happen inside `mymodel.h` (which you can name anything you like). You will also have to update your parameter and input files as the model requirements change. The parameter file can be autogenerated, something that is explained in the tutorials.

4.2 Entity registration, handles, and equation bodies

The `inca_model` object contains lists of various model entities (see Table 2.1 for an explanaition) and information about these. It is the model developers job to register all the entities and provide the needed information for each one.

Example 9. Example of entity registration.

```
auto X = RegisterInput(Model, "X");
auto A = RegisterParameterDouble(Model, System, "A", Dimensionless, 1.0);
```

The input "X" is tied to the model and is given the name "X". The parameter (of type double) is tied to the Model, the parameter group System, given the name "A", the unit Dimensionless, and the default value 1.0. A full model building API describing all the registration functions will be given later.

Each registration function returns a handle. In Example **??**, the handles are `X` and `A` (declared with `auto` to signify that we don't care about their exact type). Handles are "tickets" one can use to talk about an entity one has already registered. If you provide the handle `A` in the context of extracting a parameter value, the model will know that you are talking about the parameter "A" that you registered earlier.

Every equation in the model also receives an equation body, and inside the equation body one can reference the values of registered parameters, inputs and equations by referring to them by their handle.

Example 10. The equation "AX" references the value of the parameter "A" and the input "X" by using the handles returned from the registration of those entities.

```
auto AX = RegisterEquation(Model, "AX", Dimensionless);
EQUATION(Model, AX,
        return PARAMETER(A) * INPUT(X);
)
```

Note 3. If you are confused by the syntax of declaring an equation body because it does not look like valid C++, don't worry. `EQUATION` is a macro that expands to declaring a C++11 lambda that is then stored in the `Model` object. The macro is needed to hide some of the underlying functionality, like what object the `PARAMETER` and `INPUT` accessors read their values from, so that the model builder does not have to type too much repetitive code all the time.

4.3 Implementing a mathematical model in INCABuilder

The SimplyP model [2] uses the following very simple snow model.

Symbol	Type	Units	Description	Default	Min	Max
$D_{snow,0}$	parameter	mm	Initial snow depth	0.0	0.0	10000.0
f_{DDSM}	parameter	$mm\,d^{-1}\,°C^{-1}$	Degree-day factor for snowmelt	2.74	1.6	6
$Pptn$	input	$mm\,d^{-1}$	Precipitation			
T_{air}	input	$°C$	Air temperature			
P_{snow}	state variable	$mm\,d^{-1}$	Precipitation falling as snow			
P_{rain}	state variable	$mm\,d^{-1}$	Precipitation falling as rain			
P_{melt}	state variable	$mm\,d^{-1}$	Snow melt			
D_{snow}	state variable	mm	Snow depth (water equivalents)			
P	state variable	$mm\,d^{-1}$	Hydrological input to soil			

Table 2: SimplyP snow parameters and state variables

The state variables are given by the following equations

$$P_{snow} = \begin{cases} 0 & \text{if } T_{air} > 0 \\ Pptn & \text{otherwise.} \end{cases}$$

$$P_{rain} = \begin{cases} 0 & \text{if } T_{air} <= 0 \\ Pptn & \text{otherwise.} \end{cases}$$

$$P_{melt,t} = \min(D_{snow,t-1}, \max(0, f_{DDSM} T_{air}))$$

$$D_{snow,t} = D_{snow,t-1} - P_{melt,t}$$

$$P = P_{rain} + P_{melt}$$

Say that one also wants a different initial snow depth and degree-day factor for each reach (subcatchment). First one has to register all of the model entities.

Example 11. Declaration of model entities for the SimplyP snow module.

```
auto Mm              = RegisterUnit(Model, "mm");
auto MmPerDegreePerDay = RegisterUnit(Model, "mm/C/day");
auto MmPerDay        = RegisterUnit(Model, "mm/day");

auto Reach = RegisterIndexSetBranched(Model, "Reaches");

auto Snow = RegisterParameterGroup(Model, "Snow", Reach);

auto InitialSnowDepth      = RegisterParameterDouble(Model, Snow, "Initial␣snow␣depth"
    , Mm, 0.0, 0.0, 10000.0);
auto DegreeDayFactorSnowmelt = RegisterParameterDouble(Model, Snow, "Degree-day␣
    factor␣for␣snowmelt", MmPerDegreePerDay, 2.74, 1.6, 6.0);

auto Precipitation = RegisterInput(Model, "Precipitation");
auto AirTemperature = RegisterInput(Model, "Air␣temperature");

auto PrecipitationFallingAsSnow = RegisterEquation(Model, "Precipitation␣falling␣as␣
    snow", MmPerDay);
auto PrecipitationFallingAsRain = RegisterEquation(Model, "Precipitation␣falling␣as␣
    rain", MmPerDay);
auto PotentialDailySnowmelt  = RegisterEquation(Model, "Potential␣daily␣snowmelt",
    MmPerDay);
auto SnowMelt                = RegisterEquation(Model, "Snow␣melt", MmPerDay);
auto SnowDepth               = RegisterEquation(Model, "Snow␣depth", Mm);
auto HydrologicalInputToSoilBox = RegisterEquation(Model, "Hydrological␣input␣to␣soil
    ␣box", MmPerDay);

SetInitialValue(Model, SnowDepth, InitialSnowDepth);
```

Note that we also use the `SetInitialValue` procedure to tell the model that ”Snow depth” should have initial value ”Initial snow depth”. The value of ”Initial snow depth” is not yet determined, but it will be read from a parameter file before one runs the model. We have also added the state variable ”Potential daily snowmelt” as a partial calculation used in ”Snow melt”, however that is not strictly necessary. Next one provides the equation bodies.

Example 12. Equation bodies for the SimplyP snow module.

```
EQUATION(Model, PrecipitationFallingAsSnow,
       double precip = INPUT(Precipitation);
       return (INPUT(AirTemperature) < 0) ? precip : 0.0;
)

EQUATION(Model, PrecipitationFallingAsRain,
       double precip = INPUT(Precipitation);
       return (INPUT(AirTemperature) > 0) ? precip : 0.0;
)

EQUATION(Model, PotentialDailySnowmelt,
       return Max(0.0, PARAMETER(DegreeDayFactorSnowmelt) * INPUT(AirTemperature));
)

EQUATION(Model, SnowMelt,
       return Min(LAST_RESULT(SnowDepth), RESULT(PotentialDailySnowmelt));
)

EQUATION(Model, SnowDepth,
       return LAST_RESULT(SnowDepth) + RESULT(PrecipitationFallingAsSnow) - RESULT(
           SnowMelt);
)

EQUATION(Model, HydrologicalInputToSoilBox,
       return RESULT(SnowMelt) + RESULT(PrecipitationFallingAsRain);
)
```

In Example 12 we see that `RESULT` refers to the value of that equation for the current timestep, while `LAST_RESULT` refers to the value from the previous timestep. Looking at the model result structure that is printed using the `PrintModelResultStructure` procedure, we see the following.

Example 13. Model result structure for the SimplyP snow module.

```
[]
       -----
       Precipitation falling as snow
       Precipitation falling as rain
       -----

[Reaches]
       -----
       Potential daily snowmelt
       Snow melt
       Snow depth
       Hydrological input to soil box
       -----
```

The framework automatically determined that "Potential daily snowmelt" has to index over "Reaches" since it references the parameter "Degree-day factor for snowmelt", which we declared to be part of the "Snow" group, which indexes over "Reaches". Next, "Snow melt" does not reference any parameters, but it does reference "Potential daily snowmelt", and so it too has to index over "Reaches". Note too that even if we had declared the equations in a different order, the framework would have been able to put them in an order that makes sense. Every equation will be evaluated after any other equation that it accesses the current-timestep result of (i.e. using the `RESULT` accessor). Previous-timestep result accesses (`LAST_RESULT`) do not have any impact on evaluation order. In this case, the order of "Snow depth" and "Hydrological input to soil box" do not depend on (the current-day value of) each other, and so they could have been placed by the framework in any order respective to one another. In this example we used an input file that do not declare any index set dependencies for "Air temperature" or "Precipitation". If we instead had used an input file where these two timeseries are per reach, we would have got the following model structure.

Example 14. Model result structure for the SimplyP snow module if the user provided precipitation and air temperature timeseries per reach.

```
[Reaches]
        -----
        Precipitation falling as snow
        Precipitation falling as rain
        Potential daily snowmelt
        Snow melt
        Snow depth
        Hydrological input to soil box
        -----
```

5 Model building API

6 Application building API

References

[1] M. N. Futter, M. A. Erlandsson, D. Butterfield, P. G. Whitehead, S. K. Oni, and A. J. Wade. PERSiST: a flexible rainfall-runoff modelling toolkit for use with the INCA family of models. *Hydrol. Earth Syst. Sci.*, 18:855–873, 2014.

[2] L. A. Jackson-Blake, J. E. Sample, A. J. Wade, R. C. Helliwell, and R. A. Skeffington. Are our dynamic water quality models too complex? a comparison of a new parsimonious phosphorous model, SimplyP, and INCA-P. *Water Resources Research*, 53:5382–5399, 2017.