Documentation of Mobius framework implementation

Magnus Dahler Norling

November 1, 2019

1 Introduction

THIS DOCUMENT IS A WORK IN PROGRESS!

This document will attempt to explain some of the central concepts of the implementation of the Mobius framework. If you just want to develop or use models, you should not have to worry about this. This document will assume that the reader has understood the basics that are given in the model builder documentation, and will not re-explain concepts that are discussed there. You should also read the framework code since we will not repeat most of it here.

2 Introducing central concepts

2.1 The most important files

Almost all data structures are declared in `mobius_model.h`, along with implementation of model registration. Algorithms for finalizing the model structure and running the model are in `mobius_model.cpp`. Algorithms for organizing the storage of data (parameter values, input and result timeseries) are in mobius_data_set.cpp. These encompass the main functionality of the framework, and are also the ones that may be the hardest to understand. We will not focus that much on additional functionality such as input reading from files etc. in this document.

2.2 Entity handles

An entity handle is in practice an unsigned integer. When you register a new entity of a certain type, such as a parameter or equation, the handle of this entity will be a number uniquely identifying it. In practice, the registration information that was provided about it will be stored in the Model in an array at the index that equals the numerical value of this handle. To make it easier to ensure correctness, entity handles are typically wrapped in handle structs that just contain the numerical value in order to allow the type system to differentiate between handles of different entity types.

Example 1. Let `Model` be a `mobius_model` object. If you call

```
auto MyParameter = RegisterParameterDouble(Model, Group, "My parameter",
    Dimensionless, 0.0);
```

`MyParameter` is of type `parameter_double_h`. `parameter_double_h` has a single member, `Handle`, which is an unsigned integer. When you call the function, it will push a new element onto the `Model->Parameters.Specs` vector. This new element is a `parameter_spec`, which is a struct that among other things contains the name, type, group, default value etc. that was provided in the `RegisterParameterDouble` call. The `Handle` of the `parameter_double_h` returned is the index of this new `parameter_spec` in the `Model->Parameters.Specs` vector.

There is a separate `Model->X.Specs` vector for each entity type `X`. This is because the specs of different entity types have to store slightly different information. For each entity type the model also has a hash map `Model->X.NameToHandle` that allows you to retrieve the handle of an entity given the (string) name that it was registered with.

The purpose of returning the handle of a registered entity is that you should be able to refer to it in future registrations. This is either in registration of other entitites that are related to this one or when you look up the value of a parameter, input or equation inside the body of an equation.

Example 2. Say that this was a very simplified model building system where each parameter only had a single value and did not index over index sets. Then you could organize your parameter values into a single array `ParameterData` and put the value of a parameter with handle `Handle` at location `ParameterData[Handle]`. Then inside an `EQUATION`, the value lookup `PARAMETER(MyParameter)` could just essentially return `ParameterData[MyParameter.Handle]`. Unfortunately things are not that simple, and so the implementation is a little bit more complicated, but more about this later. This is the basic idea that the value access system is built from, and so it is important to understand this idea. The entire design of the Mobius framework was built starting with this basic idea. What this achieves is that accessing the value of a parameter is just a very cheap array lookup.

2.3 Equation bodies and the model run stater

An equation body is a (C++11) lambda function that is tied to a registered equation (i.e. to an equation handle). If evaluated, this lambda will give the value of this equation given the values of parameters, and other things the equation accesses to produce its value.

Example 3. If you during model registration provide the code

```
EQUATION(Model, MyEquationHandle,
<some code..>
)
```

the `EQUATION` macro will expand this to

```
SetEquation(Model, MyEquationHandle,
[=](model_run_state *RunState__)
{
<some code..>
}
);
```

The `SetEquation` function just stores the lambda in `Model->EquationBodies[MyEquationHandle.Handle]`.

This way the model knows what equation lambda is tied to what equation handle, and it can evaluate this lambda later during the model run.

The `model_run_state *RunState__` is an object that contains whatever information the equation needs to look up in order to find the values it needs (more on this below). The model has to set up this value set accessor with the right information when it wants to evaluate the equation lambda, and this will typically change when indexes or timesteps are changed.

Example 4. The code

```
EQUATION(Model, MyEquationHandle,
       return PARAMETER(MyParameter);
)
```

expands to

```
SetEquation(Model, MyEquationHandle,
[=](model_run_state *RunState__)
{
       return (RunState__->Running ? GetCurrentParameter(RunState__, MyParameter) :
           RegisterParameterDependency(RunState__, MyParameter));
}
);
```

We will get back to what `RunState__->Running` and `RegisterParameterDependency` do later. What is important is that when the lambda is called, the model has set up the `RunState__` object so that `GetCurrentParameter` returns the value associated to the `MyParameter` handle given the status of the index sets that the equation is evaluated under.

The reason why we use macros instead of just making the model builder type this code straight up is that we want to hide the value set accessor as an implementation detail. It is also easier to make sure the values are accessed in a correct manner this way.

Accessing values using `RESULT` `LAST_RESULT` and `INPUT` work in a similar way.

This already showcases why we wanted to wrap the entity handles in typed structs. If you call `GetCurrentParameter` on an input handle, this will make the type system of C++ give you an error, and so many mistakes can be prevented this way. Moreover, `GetCurrentParameter` is overloaded to return a different type that matches the type of the parameter (double, uint, boolean).

2.4 Dependency registration and model structure

This section is about what happens when you call `EndModelDefinition` on a model.

First it sets up a `model_run_state *RunState` with `RunState->Running = false`. For each equation handle `Handle` it will then call the lambda tied to this handle using

```
Model->EquationBodies[Handle](RunState)
```

Since `RunState->Running == false` we are in registration mode, and all the accesses the equation body does using e.g. `PARAMETER`, `INPUT` or `RESULT` will instead of extracting those values for use in the evaluation of the equation, call a registration function such as `RegisterParameterDependency`. The registration functions store information in the RunState about what accesses were made. This means that the `RunState` now contains information about what parameters, results and inputs etc. this equation accessed, and how it accessed them (implicit or explicit indexing and with what explicit indexes). This will then be used to set up the batch structure of the model.

After storing such information for every equation, we can now figure out the dependencies of every equation. First one can let each equation inherit the index set dependencies of the parameters and inputs it accesses. Then they have to inherit dependencies from other equations. This step has to be run potentially many times since index set dependencies could be inherited down many steps of equations depending on each other.

Next, the model batch structure is determined. This is a complex many-step process. What it tries to do is explained in the section "Determining the batch structure of advanced models" in the model builder documentation. That section does however not explain the implementation of the algorithm.

TODO: Give a short description of the chosen algorithm here.

Finally, `EndModelDefinition` stores some data with the batch structure that allows `RunModel` to be more efficient with how it updates the `RunState` during the main model run. More about that later.

2.5 The main value storage in the dataset

The `mobius_data_set` has a buffer that stores all the parameter data in contiguous memory. This is the `DataSet->ParameterData` array. Similarly there is a `DataSet->InputData` array and a `DataSet->ResultData` array that store all input data and result data respectively. The reason why we choose to store them in contiguous memory instead of for instance having one array for each input where we store the timeseries for that input is that we try to have a storage strategy where we avoid cache misses as much as possible.

To summarize briefly, most modern processors have 2-3 levels of memory caches (L1, L2 and L3). These speed up memory lookups of values that were stored in memory close to a value you recently looked up.

This means that if you can place things that you know are going to be accessed close together in time also close together in memory, your program can potentially get a lot faster. This is a very important optimization technique in time-critical programs. TODO: Find a good reference for this.

We have chosen a general storage structure layout. Each storage structure has a list of "storage units", where each storage unit has a list of handles and a list of index sets. The storage is organized so that the memory for each unit is placed after one another in memory. Each unit is then a multi-dimensional array. The number of dimensions is equal to the number of index sets, and are indexed by these index sets. This is an array of blocks, where each block contains a list of values that corresponds to the list of handles. TODO: make a diagram.

For parameters, there is just one such structure. For results and inputs, the entire structure is repeated once for each timestep (for results, including the "initial value step").

The necessary information needed to look up the storage index (i.e. location in memory) of a specific value is kept in the `storage_structure` struct. The actual data is not a part of this struct and is kept separately in the `mobius_data_set`, such as in the aforementioned `DataSet->ParameterData` array. There are several functions called `OffsetForHandle` that does the computation of this storage index based on what handle you are looking up the value for, how your index data is organized and so on.

TODO: More details!

2.6 More on the value set accessor system

The `model_run_state` has four buffers `CurParameters`, `CurResults`, `LastResults`, `CurInputs`. During the model run, the main model loop (`RunInnerLoop`) function will make sure that for each timestep and for each change of indexes, these buffers will be set up so that for instance for the parameter handle `H`, `CurParameters[H]` is the value of that parameter for the given indexes. This makes it so that if you use `PARAMETER(H)` to look up the value of this parameter inside an `EQUATION`, that will only be an array lookup, which is fast.

First one can think of a very simple algorithm to achieve this:

Example 5. Pseudocode for a very simple way to update all the parameter values

```
for each parameter handle H :
       memory_location = OffsetForHandle(DataSet->ParameterStorageStructure,
           RunState->CurrentIndexes, H)
       RunState->CurParameters[H] = DataSet->ParameterData[memory_location]
```

Here `RunState->CurrentIndexes` must first be updated with the current state of the index sets so that we retrieve the right values corresponding to the right indexes.

However, this would be slow for a few reasons

i We are looking up values of all the parameters, but we are probably not going to need all of them for the current equation batch.

ii The OffsetForHandle computation is relatively slow.

To address (i), remember that we actually know all the handles that are going to be accessed by each equation, because this was registered during the `EndModelDefinition` step. So at the end of `EndModelDefinition` we store with each batch group all the handles that are looked up by equations in that batch. Moreover, since some of these handles can be to entities that don't index over as many index sets as the batch itself, we also store at which index change we need to look up the value again. This means that not all values have to be loaded in again at every index change. In detail, if an equation batch group indexes over the index sets in `BatchGroup->IndexSets`, then what parameters, results and inputs to update in the `RunState` are stored in `BatchGroup->IterationData`. In pseudocode, running the batch group (See `ModelLoop` and `RunInnerLoop` in `mobius_model.cpp`.)

Example 6. Pseudocode for how one could evaluate one batch group of the model for one given timestep.

```
for each Index corresponding to the index set BatchGroup->IndexSets[0] :
       RunState->CurrentIndexes[IndexSets[0]] = Index
       update the RunState with all values specified by BatchGroup->IterationData[0]
       for each Index corresponding to the index set BatchGroup->IndexSets[1] :
              RunState->CurrentIndexes[IndexSets[1]] = Index
              update the RunState with all values specified by BatchGroup->
                  IterationData[1]
                      .
                      .
                      .
                      for each Index corresponding to the index set BatchGroup->
                          IndexSets[N] :
                             RunState->CurrentIndexes[IndexSets[N]] = Index
                             update the RunState with all values specified by
                                 BatchGroup->IterationData[N]
                             for each Batch in the BatchGroup :
                                    solve/evaluate all equations in the Batch
```

In practice the structure of this code is implemented a little differently since the number of index set dependencies per batch group is variable.

To address (ii), since we know that for each timestep, it is going to look up the same parameters in the same order as in the last timestep, we cheat and go through this process just once before we start the main model run. This is the `FastLookupSetupInnerLoop`. This gives us a list of parameter values in the order that they are going to be looked up. When we are executing the main model run, we can then just read from this list in order each timestep instead of re-computing the memory indexes and looking up the values from main storage. The lists are called `RunState->FastParameterLookup` etc.

For results and inputs it is a little more complicated: Since we have a different value for each timestep, we store a memory offset in the fast lookup list instead of a value. This is a memory offset from the start of the storage of the current timestep (or previous timestep if you are looking at last-results). Moreover, the `RunState` will always be updated by the `RunModel` procedure at each timestep to have a pointer to the start of the storage of the current (and previous) timestep. This allows for looking up the needed values.

2.7 Storage of result values

When it comes to result values from equations, the `DataSet->ResultStorageStructure` is always set up to mirror the model batch group structure. What this means in practice is that results of equations are stored in the main DataSet-¿ResultData array exactly in the order that they are evaluated by the model. This gives a small extra performance win in a few instances. For instance, when equations are evaluated/solved, the results can just be written in order into memory instead of having to also figure out at which index they are to be placed. It also allows for easier looking up of results of these equations from the previous timestep in case they are needed by a `LAST_RESULT` lookup.

TODO: Document further details for looking up and storing out result values.