Documentation of Mobius framework implementation

Magnus Dahler Norling

May 7, 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->ParameterSpecs` 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->ParameterSpecs` vector.

There is a separate `Model->XSpecs` 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->XNameToHandle` 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 be a macro that expands to `ValueSet__->ParameterData[MyParameter.Handle]`. Unfortunately things are not that simple, but this is still the basic idea that the value access system is built from, and so it is important to understand this idea. More about this later. What this achieves is that accessing the value of a parameter is just a very cheap array lookup.

2.3 Equation bodies and the value set accessor

An equation body is a (C++11) lambda function that is tied to a registered equation. 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,
[=](value_set_accessor *ValueSet__)
{
<some code..>
}
);
```

The `SetEquation` function just stores this lambda in the `Model->Equations` vector at the index `MyEquationHandle.Handle`. This way the model knows what equation lambda is tied to what equation handle, and it can evaluate this lambda later during model run.

The `value_set_accessor *ValueSet__` 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,
[=](value_set_accessor *ValueSet__)
{
       return (ValueSet__->Running ? GetParameterValue(ValueSet__, MyParameter) :
           RegisterParameterDependency(ValueSet__, MyParameter));
}
);
```

We will get back to what `ValueSet__->Running` and `RegisterParameterDependency` do later. What is important is that when the lambda is called, the model has set up the $\hat{\text{t}}$ ValueSet__ object so that `GetParameterValue` 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 of `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 `GetParameterValue` 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, `GetParameterValue` is overloaded to return a different type that matches the type of the parameter (double, uint, boolean).