The Mercury Code

Tim Larkin

version 1.0 / 16 February 2008

The Hermes suite consists of two major pieces, Hermes proper, which is used to design models, and Mercury, which is used to execute simulations of Hermes models. This document describes Mercury. It will provide a narrative of the organization of the code, without getting into coding specifics. If you want to make modifications to Mercury, this is the place to start. I assume a working knowledge of Objective-C.

Introduction

The Mercury code can conveniently be divided into two parts. One part consists of the class definitions for the components of the model that is created in the Hermes editor. The second part is a simulation engine, which initializes the model components, and then updates them to represent simulation time. The coding of the first part is the same, with some small exceptions, for both Linux and Mac. The second part will necessarily be different on the two platforms, since the characteristics of the simulations on each differ.

Although creating code of this second sort will be the immediate interest of our readers, we will start with the model components, since understanding them presents a necessary prerequisite to designing a simulation engine.

The Abstract Component

All the classes of the model derive from the HMNODE class.[1] HMNODE defines some cosmetic member variables for use in the Hermes editor. It also defines a name, which is a string.

HMNODE is the ancestor class for HMPART and HMPORT.[2] An HMPart is the ancestor class of all the components that appear as choices in the Hermes editor, such as HMTVDD, a time-varying distributed delay, and HMLOOKUP, an arbitrary function generator. It is perhaps easiest to think of each descendent class as defining a mathematical function, which takes some[3] arguments and produces some values. Each of these inputs and outputs is represented by an instance of HMINPUT or HMOUTPUT, which are subclasses of HMPORT. It will be evident that each HMPART owns a set of HMINPUT and HMOUTPUT instances.

Hermes models are hierarchical because of a special subclass of HMPART called HMLEVEL, which is a node with a set children, which are subclasses of HMPART. Since HMLEVEL is also a subclass of HMPART, the model tree can be of arbitrary depth.

Evidently each instance of an HMPORT must evaluate to a value which is either an input to a function or an output of one. Since values in Hermes are not typed, a value is represented by a C union structure named Value. The union tag supports floating point numbers, arrays of floats, and arrays of arrays or matrices.

Obviously the values of outputs are generated by the function of the component's class. The values of inputs can be set in either of two ways. By default each input has a constant value. An input which is not a constant takes its value from the output of some other component. In the simple case, where both the input's and the output's components are peers, a connection is made directly between them. This wiring never crosses directly across different branches of the model tree. Instead an input or output can be published to the parent node of its component, which produces a sort of alias to the port in the parent. By repeated publishing, an input or output can be raised up the tree until it appears at a level suitable for making a connection between peer components. Publishing produces inputs and outputs for HMLEVEL instances, which simply pass values around the tree without changing them in any way. Published inputs don't have to be connected; they can have constant values.

The Standard Class Methods

The main methods used by the simulation engine are initialize, `updateStates`, and `updateRates`.

Initialize is called once at the beginning of the simulation. Any component class uses this method to carry out instance initialization. The initialization done by a time-varying distributed delay will serve as a good example. An HMTVV owns four arrays. Three of these could be reallocated on every time step. The fourth holds the state of the delay, and so must be kept throughout the simulation. For reasons of efficiency, the other three are also allocated once and reused. The array allocation occurs during initialization. In addition, the state array can be initialized to a user supplied value. This too is done during initialization. Finally, the output that gives the total storage in the day is initialized.

Once initialization is complete, all the outputs of the model have the values that define the state of the simulation at time zero. From this state, we can compute the rates that are assumed to hold until *time += Δtime*. This is done by calling `updateRates` on each of the components.[4] The values of state variables are invariant during this phase. In the HMTVDD, coefficients are computed for the development rate, the loss rate, and the seepage rate. These coefficients are multiplied by each element of the storage array, and the products are stored in three rate arrays.

Next states are updated. The values of states depend only on the values of rates, never on the value of other states. Consequently states can be updated in any order. As a matter of fact, they are updated in compliance with the dependency graph. In

1 "HM" is the universal prefix for the component classes in both Mercury and Hermes.

2 The part-port distinction is a somewhat unfortunate choice of nomenclature since the words are more similar than different. However, it's too deeply embedded now to be worth changing.

3 Or none.

4 We have glossed over the problem of the order in which we update the rates. Some rates may be computationally dependent on other rates, and so must be updated later. This dependency graph is calculated at a much earlier state, when the components are instantiated, and based on the graph, the components are ordered into an queue that obeys the graph's constraints.

the HMTVDD, the state array is incremented by the sum of the rate arrays times *Δtime*.

At this point, the states have been updated to values that represent the simulation at the beginning of the next time step. But before time is advanced, the outputs identified as the response variables of interest get an opportunity to write their values to their files.[5]

Finally it's time to go back to the top of the loop, increment time, and start all over again.

Writing a simulation engine

Now we can get into the nitty-gritty of writing a simulation engine.

At the top level is the standard C main routine, which sets up the auto-release pool, reads `main's` argument, which is a path to the setup file, creates an instance of APPCONTROLLER, and sends it the `runsimulation:` message with the path as an argument. This is pretty basic, and probably won't need to be changed. The runsimulation method does three tasks. It calls `getRunParameters` to read the setup file, and sends the top-most stepper the `startWithPath:andInferiors:` message. When this returns, the simulation is done, and before exiting, the APPCONTROLLER closes all output files.

I didn't tell you about steppers? The evaluation of a model typically involves running it with a range of values for one or more interesting parameters. To use one simple example, you might want to see the effect on an HMTVDD of changing the value of its shape parameter. To answer this need, I created the HMSTEPPER class. A stepper knows 3 basic numbers, a start value, an increment value, and an end value. Steppers are typically nested, since there is typically more than one model parameter of interest, so each stepper also knows its inferior. And it has a list of the instances of HMINPUT which correspond with the parameter.[6] It responds to the `startWithPath:andInferiors` message by initializing its value to the start value. It then runs a loop, where it applies the value to each element in its HMINPUT list, sends `startWithPath:andInferiors:` to its stepper inferior, updates its value by its increment, and checks to see whether its value has exceeded the end value. If not, the loop repeats; otherwise it ends and the method completes.[7]

At the bottom of the stepper list (or the top if there are no other steppers) sits an instance of HMCLOCK, a subclass of HMSTEPPER. A clock runs the entire simulation, from time zero until the end. Like the other steppers, it has a start value, an end value, and an increment, but these refer not to model parameters, but to simulation iterations. Instead of calling an inferior stepper within its loop, it calls on the APPCONTROLLER to run the simulation for one iteration. The APPCONTROLLER does this by updating each of the models in the simulation space by one iteration of the procedure described in the previous section.

The consequence of this design is that all you need to do to execute a simulation over all the simulation space and all the parameter space, is to send the top stepper the `startWithPath:andInferiors:` message. If I have programmed the thing correctly, you can send this message to the top stepper as many times as you like.

Producing output

There are currently two ways of identifying an output to write its current value to a file. The original approach relied on adding appropriate elements to the setup XML. The newer approach was developed to provide a way to do this within the Hermes editor. I will describe the second approach first, since that is the only one currently in use.

The HMOUTPUT class adds a boolean instance variable named `recordP`. The user can set or clear this variable using a menu.

In this approach, all the output is written to a single file. But first, after the model is instantiated but before the simulation begins, `setupOutputsFromRecorders:` is called to collect the outputs, and their names are concatenated into a string. The names of all the steppers are accumulated and prefixed to this string. This entire string is written to the output file as the column headers. During the simulation, the recording outputs write their values as tab-delimited text to a single file. Each row in the file represents the values at the end of a single time step. In front of each row we write the current values of the steppers. This list is accumulated and passed down the stepper hierarchy in the first argument to `startWithPath:andInferiors:`.

In the original approach, each output is associated with a different file, the path of which is specified in the setup XML. The APPCONTROLLER makes a list of file pointers which is ordered to parallel its list of recording outputs. When the output receives the `recordValue:` message, it also gets its appropriate file pointer. By redesigning this file pointer list, many variations can be attained.

The situation becomes a bit more complex in a spacial simulation. In this case, all the instances of a specific recording output write their values to the same file. An output in a 10 x 10 space writes 100 values per row, prefixed by the set of stepper values. It would not be hard to modify the program to write all values to a single file even here. Then a model with 10 response variables would result in each row containing 1000 numbers plus the stepper values. Obviously this would be impossible to read directly.

Reading files

The current approach of Mercury to reading files is quite primitive, since it was designed at a time when there were no specifications of how the weather file would be structured. In any event, the task is handled by instances of HMFILE. These instances have a parameter that defines a path to a file. During initialization, the file is opened. During `updateRates` the first value is read from the file, which becomes the value that the component outputs. Then the file pointer is advanced to the end of the current line, so that regardless of how many values are on a line, the next one to be read will be the first value of the next line.

If EOF is read, an exception is raised which brings the simulation to an end. But this exception is caught by the APPCONTROLLER, so that all output files are properly closed.

Obviously this is a weak solution which is suitable to only the simplest cases, and it will need to be improved as the cases become more realistic, and the file specifications become more certain.

5 The groundwork for this is done when the model is being instantiated, when the response variables, which are instances of HMOUTPUT, were collected into an array.

6 There can be more than one such input because there can be more than one instantiation of a model in a simulation.

7 The list of steppers is created when the setup file is read. The writer of an engine can assume that the APPCONTROLLER instance variable steppers contains all the steppers.