HOW does the OpenMI Linkable Component calling sequence work?

Phases in using the linkable component interface

OpenMI systems are composed of OpenMI components that can be used in specific ways. The OpenMI standard identifies a number of phases in the use of OpenMI linkable component. Figure 1 provides an overview of the phases that can be identified and the methods that might be invoked at each phase. While the sequence of phases is prescribed, the sequence of calls within each phase is not fixed.

Figure 1 Deployment phases of OpenMI linkable components
The dynamic behaviour of the various phases is discussed in more detail below.

Phase I: Instantiation and initialization

This phase is the entry point of the process to establish an OpenMI system. At the end of the phase, a linkable component has sufficient knowledge to populate itself with model data and expose its exchange items. Whether the linkable component has been populated with model data depends on the solution chosen by the code developer.

To instantiate and prepare the river and groundwater model combination the following steps are needed (Figure 2):

- **Instantiation**: In this phase the application reads the OMI file, which refers to the software unit (i.e. assembly) that implements the LinkableComponent. Using this reference the LinkableComponent will be constructed.
- **Initialization**: The LinkableComponent can be populated with input data by calling the `Initialize()` method with the arguments as listed in the OMI file. The arguments typically contain references to data files.
Phase II: Inspection and configuration

At the end of this phase, the links will have been defined and added and each component will have validated its status. In some cases, this phase might be very straightforward; this is the case for a hard-coded system, for example. In an 'open' system, however, the examination of available exchange items plays a crucial role. The simplest way to retrieve exchange items is to ask for the number of exchange items and loop through the list. The providing component can only implement this directly if the exchange items are static and known a-priori. In those cases where exchange items are a-priori unknown (i.e. they depend on the connected components) a dynamic query process will take place. Figure 3 shows how requests for exchange items are resolved (for example, by examining other components for their exchange items).
Once the ExchangeItems have been examined the links can be prepared and added to the component. Figure 4 illustrates this process for the rainfall-runoff/river model example.

**Figure 4 Adding links in the rainfall-runoff/river use case (sequence diagram)**

1. Create the relevant objects (links, quantities, element sets, data operations). Check the validity of selected data operations and populate the objects.
2. Add the links between the components:
   - **LinkRainToCatchm** (links rainfall to the catchment): The target component (RRmodel) needs to know from which component to obtain the rainfall. The source component (RainModule) needs to know the target element set where it has to deliver the rainfall.
   - **LinkRunoffToInflow**: The target component (RiverModel) needs to know from which component to obtain the lateral inflow. The source component (RRmodel) needs to know the target element set where it has to deliver lateral inflows. Note that a flux towards the target destination (RiverModel) is positive.
   - **TriggerLink**: A trigger link to the river model - the downstream component in the data chain - is created and populated. The purpose of this link is to enable the first GetValues() call to the RiverModel. This call triggers the calculation chain.
3. Validate the status of the components and their links using the Validate() method.

**Phase III: Preparation**

This phase is entered just before the computation/data retrieval process starts. Its main purpose is to define a clear take-off position before the bulky workload starts. This phase contains only one method: Prepare().

During this phase database or network connections (or both) might be established, monitoring stations might be called or model engines might prepare themselves by populating themselves with schematization input data (if this has not been done before), opening their output files, organizing their buffers, creating their data mapping matrices for (spatial) interpolation purposes, etc.

Note that this phase must include a final validation of the status of the linkable component.

**Phase IV: Computation/execution**

During this phase, the heavy workload will be executed and associated data transfer will get bulky. The data transfer mechanism of OpenMI is defined as a request-reply service mechanism, having direct interaction between two linkable components without any involvement of external facilities. Section 3.3 of the standard explains how data exchange takes place for unidirectional data exchange, bidirectional data exchange and iteration.

The computation starts by invoking the GetValues method via a link. This link can be connected to a trigger object (an 'empty' implementation of a LinkableComponent) or it might be connected to 'real' components such as a visualization tool or an output file.

While OpenMI linkable components sort out their own data communication and time synchronization, system developers should still pay attention to the question of process control.

System developers can easily develop an application that invokes the component chain with the end time as an argument in its GetValues call. The consequence is, however, that the model components do not return control to the main application until they are finished, thus making any smooth interruption of the process difficult. In addition, the 'overall' step size of the computation remains 'out-of-sight' as it is decided by an unknown component.

Given those disadvantages it is therefore recommended that you implement some execution facility that turns the entire simulation period into a loop of timesteps. (Within the org.OpenMI.Configuration package this execution facility is called the deployer.) While progressing through its simulation period, the GetValues call will be invoked for each step. Between each call, the application may react to external events, while the execution facility keeps control over the process. Figure 3 7 illustrates the function of such a loop.
try {
    MyRainModule.Prepare();
    MyRRModel.Prepare();
    MyRiverModel.Prepare();
    int nrsteps=100;
    DateTime step = (end - start)/nrsteps;
    DateTime stop = end + 0.000001;
    DateTime _time = start;
    while (_time <= stop) {
        Application.DoEvents();
        IValueSet Values = _trigger.GetValues(new TimeStamp(_time),
                                               _triggerLink.ID);
        _time = _time.AddSeconds(step);
    }
    GermanRhineModel.Finish();
    NethRhineModel.Finish();
} // end try

Figure 3 7 A time loop

Phase V: Completion

This phase comes directly after the computation/data retrieval process is completed. Code developers can use this phase to close their files and network connections, clean up memory etc. This phase contains only one step with one method-call: Finish.

Phase VI: Disposal

This phase is entered at the moment an application is closed. All remaining objects are cleaned and all memory (of unmanaged code) is de-allocated. Code developers are not forced to accommodate re-initialization of a linkable component after Dispose has been called.