Climate-KIC

BGD Integrated modelling system

Authors: Evangelos Rozos, Stefanos Kozanis and Christos Makropoulos

Table of Contents

Introduction

BGD is about interactions between different physical domains, between biological and engineering systems, between technologies and social systems, etc. One approach to study this hyper-system could be to develop from scratch a monolithic hyper-model that would simulate every aspect of this conceptualization. However, this choice, apart from being a resource-intensive task (money spent for development, and time required for validation and verification), would also lack of flexibility and openness (difficult to include new features) and would have to cope with the diversity of those involved. Instead of employing an every-aspect modelling, a more elegant approach, which could turn the diversity into an advantage, would be to set-up a mechanism that would capture the interactions between the physical domains. Then, the hyper-system could be broken down to sub-domains, which could be modelled independently, but still interacting between each other ensuring thus an integrated modelling.

The capture of interactions includes the interoperability, communication and exchange of data (possibly in real/run time) between different tools developed by different people. This can be accomplished by adopting standard interfaces, which will allow the developers to progress with their work “under the hood” independently. A key advantage is that each tool developed using the common standards will be able to talk not only to other BGD tools/models, but also to everything else that complies to these standards (USEPA, ESRI, DHI, DELTARES, MWHSoft, etc).

The use of standard interface between models belongs, according to Brandmeyer and Karimi (2000), to the highest level of the methodologies used for model coupling, where models are coupled using an overall modelling framework. A modelling framework that lately becomes more and more popular is the OpenMI (Open Modeling Interface), which has been developed with the purpose of being the glue that can link together model components from various origins. OpenMI provides a standardized interface to define, describe and transfer data on a time basis between software components that run simultaneously. This report provides guidelines to help with the implementation of this interface.

The OpenMI standard

OpenMI allows the linking of models with different spatial and temporal representations: for example, linking river models and groundwater models, where the river model typically uses a one-dimensional grid and a short timestep and the groundwater model uses a two- or three-dimensional grid and a longer timestep (Gregersen et al., 2007). The OpenMI 2.0 Standard (latest release) is followed by the release of the FluidEarth. FluidEarth (the current stable version – February 2014 – is 2.0, but RC1 of 2.2 has been released and soon will become the official stable version) is an open source implementation of the OpenMI 2.0 interface standard for data exchange between numerical models during run-time. It consists of the FluidEarth Software Development Kit (SDK) to assist with making models and components OpenMI 2.0 compliant, and the Pipistrelle GUI for linking and running OpenMI compliant models and components using one-way or two-way links for the exchange of data (Sutherland et al., 2013).

OpenMI is designed to accommodate the easy migration of existing modelling systems since their re-implementation may not be economically feasible due to the large investments that have been put into the development and testing of these systems. For legacy code, wrapping will most often be the technological choice to migrate to the OpenMI. An existing model engine (i.e. a computational core, developed in Fortran or C or any other language that produces native code) is encapsulated in a so-called wrapper that meets the interface specification of an OpenMI linkable component. The OpenMI interface allows it to be treated in a generic way by an OpenMI run-time environment. Actually, the wrapper turns the computational core into a linkable component for the OpenMI (Moore et al., 2010).

OpenMI can also incorporate scripting languages commonly used for modelling and analysis (e.g. MATLAB, Scilab, and Python). Bulatewicz et al. (2013) have developed a general purpose software component for the OpenMI that simplifies the linking of scripted models to other components. The Simple Script Wrapper (SSW) provides a foundation for model linkages and integrated studies. This solution enables scientists to easily make their scripting language code linkable to OpenMI-compliant models. The SSW is available online as supplementary material to the publication of Bulatewicz et al. (2013), which also includes an example that may be followed to employ their method.

OpenMI compositions

An OpenMI composition is a set of components linked and run together, exchanging data as they run. The figure below displays a composition with two OpenMI components.

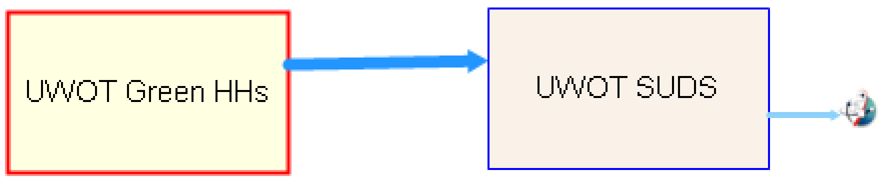

Figure 1: Example of an OpenMI composition that includes two UWOT components.

The left component of Figure 1 (named “UWOT Green HHs”) is a UWOT (Rozos and Makropoulos, 2013; Rozos et al. 2013) component that simulates the water cycle of a group of identical households that employ Green technologies (see Figure 3), whereas the other component (named “UWOT SUDS”) simulates a central SUDS (see Figure 4). The “UWOT SUDS” receives the runoff output of the “UWOT Green HHs” component. The output of the “UWOT SUDS” is connected to a "Get Values Call" component (this serves as the terminal of a composition), which indicates to the composition solver the simulation starting point (the components are run starting from the component directly connected to the "Get Values Call" and proceeding upstream). It should be noted that both UWOT components in this composition use the same wrapper, but have different OMI files.

Examples on how to prepare an OpenMI composition using Pipistrelle can be found in the following videos:
http://elearning.fluidearth.net/video/comp_p1_simple/comp_p1_simple.swf
http://elearning.fluidearth.net/video/comp_p1_double/comp_p1_double.swf

How to make existing models OpenMI compliant

To make existing models OpenMI compliant there are two options. The first one is to add the engine's native code (for difference between native and managed code see the section Connect the wrapper to the engine) to the same solution that includes the project of the wrapper. Then, use the SafeNativeMethods class to call the engine. See and example in these videos:

http://tinyurl.com/Build-simple-component-1
http://tinyurl.com/Build-simple-component-2
and in this code:
http://tinyurl.com/Build-simple-component-code

However, this option requires the Professional (non-free) version of Visual Studio and is not applicable to every programming language.

The second option is to use an independent compiler to build the DLL of the engine. The required steps are:

1. Prepare the DLL that contains the core engine of the model.
2. Prepare the OpenMI wrapper, i.e. the C# classes that wrap the engine model and implement the keys required by the OpenMI standard.
3. Connect the wrapper to the engine.

For the easier integration of the BGD components, the wrapper should ensure that the units of the outputs should comply with those defined in the following table or (in case of a unit that is required but does not appear in the following table) with the SI convention.

Table: Recommended units for inputs/outputs of BGD components.

Type (example)	Unit
Flow (runoff)	m^3/s
Areal stress (rainfall)	mm/s
Energy	kWh
Temperature	Celsius

Likewise, the wrapper is responsible for converting the units of the component inputs to those required by the wrapped engine.

Prepare the core engine

If the model is to be built from scratch, then it would be better to implement it directly in a C# class, and call it from the method Update, as it is defined by the OpenMI standard. However, in the case of a model already developed in another programming language it would require a lot of resources to rewrite it in C#. Happily, any model can be used in an OpenMI composition provided that its engine core is compiled into a shared library/a dynamic link library file (DLL).

Ideally, we should make minor modifications to the engines so that the same engine core can be used both when running in the OpenMI environment and when running as a standalone application. The preferable approach is to make a new application (EXE) that calls a function in the engine core DLL that, in turn, makes the engine perform a full simulation (OATC, 2010).

The engine, when implemented in a DLL, can be wrapped in a C# assembly that implements the key classes required by the OpenMI Standard. This assembly is called the wrapper, which, in turn, is compiled in another DLL. This DLL can be inserted into an OpenMI composition to link the model engine to the rest models of the composition. Both DLLs (engine and wrapper) should be on the same folder.

As far as concerns the wrapper, it should be written in C# and should be compiled into a DLL using Visual Studio (Mono CSharp Compiler, mcs, can be used too). As far as concerns the engine, the appropriate compiler (depending on the language) should be used to prepare the engine DLL. In the case of UWOT, of which the engine is written in C, the MinGW compiler was used to build the DLL:

- gcc -O2 -c -DWIN32 simulation.c components.c
- gcc -shared -o UWOT.dll simulation.o components.o -Wl,--out-implib,uwot_dll.a

A description of how to build a DLL from C source using Visual C++ can be found here: http://nawatt.com/index.php/corporate/blog/78-using-c-dlls-in-c (see Step 1). Note that the functions that the DLL exports should be declared in the source code with __declspec(dllexport).

Prepare the wrapper

The OpenMI Standard demands that each compliant model should implement two specific classes.
The first class, which inherits from BaseComponentTimeWithEngine, provides the description of the component (inputs, outputs and parameters) and the constructor method (the instance creator i.e. the function that allocates the memory to hold all required information). This class must implement the following public members:

- Identifier. This is an enumerator that holds the names of all arguments, inputs and outputs.
- GetIdentity. This method returns information about the parameters (called arguments in OpenMI) of the engine.
- Class constructor. The constructor is responsible for creating the argument objects for the component and adding them to the component's Arguments list.
- DoInitialise. This method creates and initialises the inputs and outputs of the OpenMI component.

The second class, which inherits from BaseEngineTime, performs the wrapping i.e. the data relay from other components to the core-engine and vice versa. This class must implement the following public members:

- Initialise. This method initialises the Arguments list.
- Prepare. This method allocates and initialises the data structure that holds the state variables of the engine.
- Update. This method calls the core engine.
- SetDoubles. This method reads the component inputs and passes the values to the engine.
- GetDoubles. This method writes the engine results to the component outputs.
- Finish. This method finalises the class (closes streams, frees memory, etc).

The OpenMI Standard is implemented in C#/.NET. To compile the C# engine-wrapper the MS Visual Studio must be installed (the Express edition is free and available for downloading from www.visualstudio.com).

The FluidEarth SDK package should be downloaded and installed (version 2.2 available from http://sourceforge.net/projects/fluidearth) to provide with the references to the methods required to implement the OpenMI interface. Detailed instructions on how to prepare a Visual Studio solution that builds an OpenMI component can be found in http://elearning.fluidearth.net.

Connect the wrapper to the engine

The link of the wrapper to the engine (i.e. data exchange and interoperability) is not trivial to accomplish because, despite the fact that both wrapper and engine are DLLs, the former is managed code whereas the latter is native code. Native code is compiled to work directly with the OS. Managed code however, is precompiled (bytecode in Java-speak) but is then processed by the Just In Time Compiler to native code at runtime. Managed code can run on different operating systems because the machine code is not created until the VM actually uses it. This way, the same wrapper DLL can be used on Windows, Linux or Mac that have the Mono (Mono is the open source equivalent of .NET) runtime installed (StackOverflow, 2014). However, the engine, if written in unmanaged code, is compiled to run for the designated OS/Hardware (this is why it is called “native” code).

In the case where a DLL derived from a native language (like C or Fortran) is called from the .NET DLL, we have mixed code (both managed and unmanaged). A description of how to import the engine functions from DLL into a C# project can be found here (see Step 2): http://nawatt.com/index.php/corporate/blog/78-using-c-dlls-in-c

In mixed code, an important difference between managed and unmanaged code is that in managed code the memory is allocated and freed automatically (see garbage collection in computer science) and the memory manager looks ahead to prevent any memory corruption (attempts to write beyond the allocated space results in an exception). On the other hand, in unmanaged code, where pointers are often used to access data, it is the responsibility of the developer to avoid any buffer overflow, which will result in memory corruption (attempting to write beyond the allocated space results either in unpredictable behaviour or in program crash).

Some examples on how to pass arguments from managed to unmanaged code and vice versa can be found in the following links:

http://stackoverflow.com/questions/2415017/convert-from-double-array-to-pointer
http://stackoverflow.com/questions/985646/c-convert-generic-pointer-to-array
http://stackoverflow.com/questions/2648560/allocating-unmanaged-memory-in-c-sharp

The OMI file

The information required to run an OpenMI component is provided by an OMI file. OMI files contain the arguments, the initial values, and the topology of a specific application of a model. This allows the same OpenMI wrapper, which is prepared by the developer, to be used with different settings and configurations described by different OMI files, which are prepared by the user (ideally using a tool provided by the developer).

OMI files are XML files of which the main element is called "Argument". Each "Argument" has two attributes, the "Key", which provides the name of the "Argument", and the "Value", which provides the value. An example of an OMI file can be found here:

http://elearning.fluidearth.net/CSharp/Part%20II/CS_ANewOMI.aspx

An OMI file may include data like real or integer numbers, booleans or strings. If more complex data types, like vectors or arrays, are required, then there are two options. The first is to include this information in an external file and then include the file’s path in the OMI file. The second option is to represent an array (or vector) inside the OMI file as a string of delimited numbers. This sequence should be tokenized by the wrapper and reshaped (reproduce a 2D array from the 1-row representation in the OMI file). The latter approach, which was used in UWOT, has the advantage that no external files are required and, hence, the risk to end up with broken links is eliminated.

FluidEarth suite comes with the tool Conformance Test, which helps test integrity and conformance of the OMI files. To use this tool you should first install the appropriate extension (filename of installation file has the form OATC_ConformanceTests_*_x86.msi). Then, access this tool either from within Pipistrelle (View → Tabs → Add → OATC_ConformanceTests) or directly from the Start menu of Windows.

How to prepare an adapter

In those compositions where the output from one component needs to go to the input of another component and the output does not match the input in spatial or temporal terms, the output can be modified by an adapter (hence "Adapted Output") so that it matches the required input (via interpolation or some other process). Some interpolations are provided as part of the FluidEarth SDK, but adaptor is for allowing developers to easily design and build their own. In much the same way that components are stored in a component library, adapters are produced when required from an adapter "factory".

An example where an adaptor is required is when the output of one component has different units from the input of the other component (e.g. L/s and m^3/s). Adapters are also useful in cases of spatial discretization. For example, the following figure displays two components named Pond #1 and Pond #2. The output of Pond #1 has 10 nodes whereas the input of Pond #2 has 5 nodes.

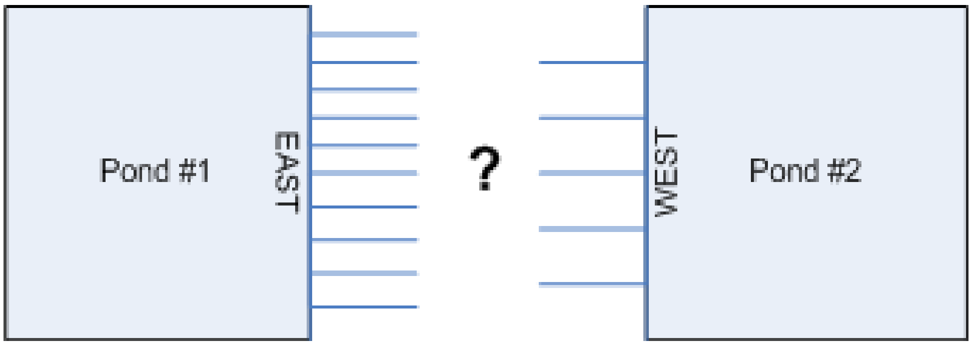

Figure 2: Need for an adaptor to convert from a 10 nodes output to a 5 nodes input (elearning.fluidearth.net/)

More information on how to prepare an adaptor can be found in http://elearning.fluidearth.net/.

An example of an OpenMI compliant model

Following the previous steps the UWOT engine was prepared in DLL (UWOT.dll), the wrapper that renders UWOT OpenMI compliant was also prepared in a DLL (UWOT_Wrapper.dll). These two DLLs are supported by a tool that helps prepare OMI files (BuildOMI.exe). The source of the wrapper and the tool can be found in the following repository:

https://vrozos@bitbucket.org/vrozos/uwot-openmi-wrapper.

The compiled OpenMI Wrapper of UWOT along with the demo composition shown in Figure 1 can be found in https://www.dropbox.com/s/wrp1rm5crvag0dg/UWOT_OpenMI.zip.

The steps that were followed to prepare this composition with the two UWOT OpenMI components were:

1. Prepare the two corresponding UWOT projects (displayed in Figure 3 and Figure 4) using the UWOT CAD application.
2. Export the UWOT project using the Export files feature (File → Export → Export files) to different folder for each project. This action exports the information of a UWOT project in a series of text files.
3. Use the BuildOMI tool (one time for each UWOT project) to compile the exported UWOT files into an OMI file (the two projects result in two OMI files, HHs_Green.omi and SUDS.omi).
4. Place the two OMI files into the same folder with UWOT_Wrapper.dll and UWOT.dll.
5. From within Pipistrelle, Right click → Add → Component → navigate to the folder where OMI files and DLLs are and click Open.

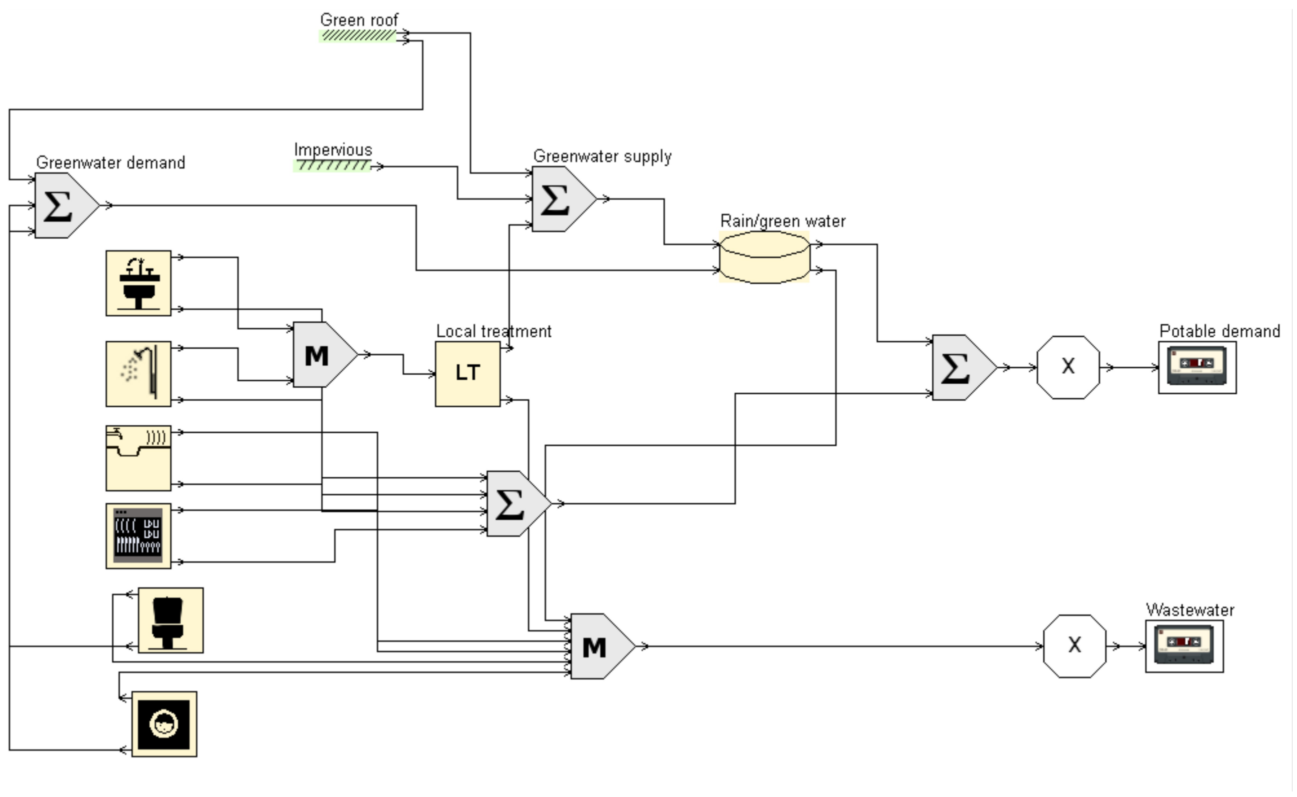

Figure 3: UWOT representation of a group of identical households with BG technologies

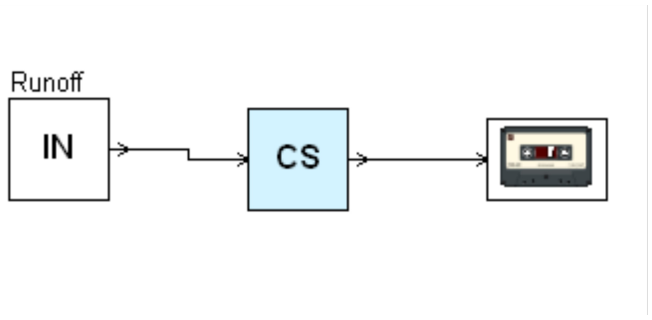

Figure 4: UWOT representation of central SUDS.

The IN component in Figure 4 is fed with the data arriving at the input of the "UWOT SUDS" component of Figure 1, and the two loggers in Figure 3 (depicted with a cassette-tape) provide with data the output of the "UWOT Green HHs" component of Figure 1.

The Pipistrelle GUI (installed with FluidEarth) provides a GIS viewer to visualize the spatial relationships between the components. Figure 5 shows the Pipistrelle GIS (View → Tabs → Add → FluiedEarth_SpatialView) displaying spatially the composition of Figure 1.

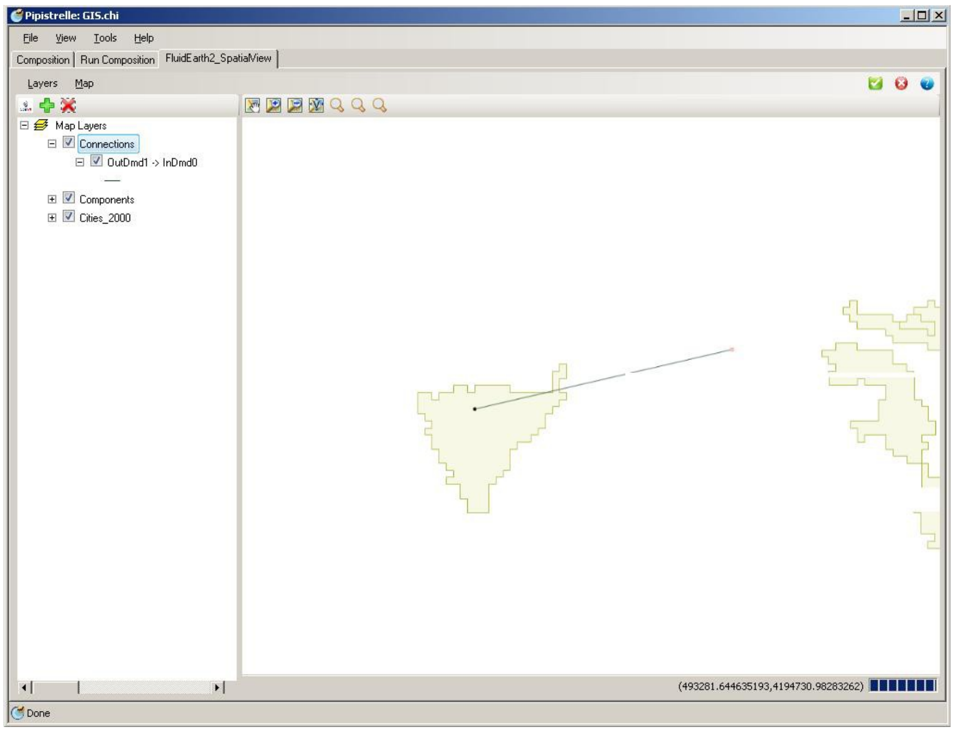

Figure 5: Spatial representation of the connection between the components of Figure 1 composition (the components are represented with two points and the connection between them with a line).

The previous description is based on a UWOT application but can serve as an indicative example for any model.

Troubleshooting

First make sure that the following major sources of problems are not your case:

- The OMI file is corrupted. If a component fails to be inserted in a composition first try to open the OMI file in an internet browser (just drug and drop into a browser window). OMI files are xml files and, if properly formed, they will be rendered by an internet browser. Check also the OMI using the Conformance Test tool.
- The OMI file is orphan. Make sure that OMI file, the wrapper DLL and the engine DLL are all on the same path.
- Multiple uses of the same wrapper. Please note that if you have a composition which uses two instances of the same component and your component is based on an unmanaged native language, like Fortran or C, then you must set the Remoting property of each instance of the component to run out of process (i.e. set the value to ipcAuto).
- Different versions. Make sure the wrapper of a model has been prepared with the same SDK version with the one used to run a composition that includes this model (i.e. do not mix the FluidEarth version 2.2 with the 2.0 or any else).

If you have run through the list above and still have problems in inserting or running a composition, then check xml and html error logs.

- An error message like "Contract.Requires(componentType != null ... Could not load file or assembly ..." indicates that the wrapper DLL cannot be found. Make sure that the wrapper and engine DLLs are on the same path with the OMI file.
- An error message like "Cannot find component argument(s) from OMI Key(s)" indicates that the name of an argument in the OMI file cannot be matched with any of the names defined in GetIdentity method. You will need to carefully check you code or edit the OMI file.
- An error message like "Sequence contains more than one element" indicates that an argument in the constructor method of the component has been defined more than once. You will need to carefully check you code.
- An error messages like "Failed component.Prepare() as component.Status != LinkableComponentStatus" indicates that an exception happened during the execution of a component. First, try to reload the project before running it (press Load in the Run Composition tab). If no success, you will need to debug you code.
- An error message like "Trying to progress engine beyond its time horizon" indicates that the date defined in the Run Options of the Get Values Call is later than or the same with the date defined in the Time Horizon of a component.
- An error message like "Object reference not set to an instance of an object." may indicate that you have more than one component that use the same engine derived from unmanaged code. In this case, you should set the Remoting argument as IPCAuto. If this is not what causes this problem, then an object in your code is used without prior initialization. You will need to debug you code.

References

- Brandmeyer, E., and Karimi, H.A., (2000), Coupling methodologies for environmental models, Environmental Modelling & Software 15 (2000) 479–488.
- Bulatewicz, T., Allen, A., Peterson, J.M., Staggenborg, S., Welch, S.M., and Steward, D.R., (2013), The Simple Script Wrapper for OpenMI: Enabling interdisciplinary modeling, Environmental Modelling & Software 39 (2013) 283–294.
- Gregersen, J.P., Gijsbers, P.J.A. and Westen S.J.P., 2007. OpenMI: Open modelling Interface. Journal of Hydroinformatics, 9(3), 175-191. http://dx.doi.org/10.2166/hydro.2007.023.
- Moore et al., (2010), Scope For the OpenMI (Version 2.0), The OpenMI Document Series.
- OATC, (2010), Migrating Models For the OpenMI (Version 2.0), The OpenMI Document Series.
- Rozos, E., Makropoulos, C., and Maksimovic, C., (2013), Rethinking urban areas: an example of an integrated blue-green approach, Water Science and Technology: Water Supply, doi:10.2166/ws.2013.140.
- Rozos, E., and Makropoulos, C., (2013), Source to tap urban water cycle modelling, Environmental Modelling and Software, 41, 139–150, doi:10.1016/j.envsoft.2012.11.015, Elsevier, 1-March-2013.

- Stackoverflow (2014), Difference between native and managed code?, available from http://stackoverflow.com/questions/855756/difference-between-native-and-managed-code.
- Sutherland, J., Bolster, M., and Haper, A., (2013), Beachplan as an Open-MI composition, Proceedings of 2013 IAHR World Congress.