The FluidEarth 2 implementation of OpenMI 2.0

Quillon Harpham, Paul Cleverley and David Kelly

ABSTRACT

Following the release of the OpenMI 2.0 standard for model coupling with reference object classes (interfaces) in C# and Java, a set of tools including a Software Development Kit (SDK) and Graphical User Interface (GUI) is expected to accompany it. These are necessary to enable numerical model developers to easily adapt their models to become OpenMI compliant and to allow modellers to easily assemble and run compositions of them. FluidEarth 2 is an HR Wallingford initiative providing these open source tools for the .net 4.0 Framework together with training, community support and sample models. They are the only such open source tools available so in this sense they act as the reference SDK and GUI for OpenMI 2.0 with .net. The purpose of this paper is to outline these and demonstrate a set of examples. A series of components were successfully constructed and compositions built. These include training models designed to demonstrate different aspects of model coupling, moving to industry strength model codes simulating dam-break bathymetry updates. The FluidEarth 2 tools have been designed to be cross-platform and have been tested under Windows and Linux (using Mono). Usage is successfully demonstrated, providing an environment for integrated modelling with OpenMI 2.0.

Key words | environment, hydraulic, integrated, interface, modelling, OpenMI

Quillon Harpham (corresponding author)
Paul Cleverley
David Kelly
HR Wallingford,
Howbery Park,
Wallingford,
Oxfordshire,
OX10 8BA,
United Kingdom
E-mail: *q.harpham@hrwallingford.co.uk*

ACRONYMS AND ABBREVIATIONS

2DH	2-Dimensional Horizontal
BIA	Bilinear Interpolation Adaptor
GIS	Geographic Information System
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IPC	Interprocess Communication
NLSW	Non-linear Shallow Water
OA	OpenMI Association
OpenMI	Open Modelling Interface
SDK	Software Development Kit
TCP	Transmission Control Protocol
UML	Unified Modelling Language

INTRODUCTION

OpenMI (Open Modelling Interface) is a standard for coupling numerical models with data exchanged between modelling components at run time. Following the successful version 1.4, version 2.0 of OpenMI was released in December 2010. The standard consists of a set of object interfaces, which can be represented by a set of Unified Modelling Language (UML) diagrams. The governing OpenMI Association (OA) also produces two sets of reference classes in C# and Java. Developers may construct their own classes from the UML diagrams, but use of the reference classes is encouraged. Following release of the base OpenMI standard, the OA also pledges to release two tools to allow the standard to be used more easily: a Software Development Kit (SDK) to assist construction of OpenMI compliant components and a Graphical User Interface (GUI) to assist the assembly and execution of compositions of OpenMI compliant components. Version 1.4 of OpenMI was accompanied by an associated SDK and GUI for C#, similar applications were also required for version 2.0 of OpenMI.

doi: 10.2166/hydro.2013.190

FluidEarth 2 is an HR Wallingford initiative providing an SDK and GUI for OpenMI 2.0 for the Microsoft .net Framework with C#. They use the C# OpenMI reference classes and are the only such open source tools available so in this sense they act as the reference SDK and GUI for OpenMI 2.0 with the .net Framework. Accompanying these tools is an extensive training website and a set of example models together with a portal for community interaction. The purpose of this paper is to outline the FluidEarth 2 SDK and GUI and, with reference to the training material, document a set of examples introducing the reader to using OpenMI 2.0 with FluidEarth 2. Although not restricted to hydrology and environmental modelling, FluidEarth 2 is driven from these disciplines and the examples listed all derive from this subject area.

Motivation

It is becoming increasingly recognised that many modern environmental questions cannot be answered by modelling physical, chemical or biological parameters in isolation. Environmental systems couple many natural processes and simulating them accurately demands modelling them in a similar fashion, that is, taking into account their interactions (Moore 2010). The environment is an interconnected system and what happens in one location can have repercussions both far away (Meiburg 2008) or at a single location if multiple, dependent parameters interact. This being the case, the only way to successfully answer these questions is to employ integrative approaches, often spanning disciplines, to complement the traditional single discipline methods (Anastas 2010).

In recognising that the actions of these complex environmental systems often produce dramatic and severe consequences, it is clear that one single numerical model cannot be sufficient to represent all of the details needed for decision making and planning (Voinov 2010). Incorporating all necessary environmental processes in a single model eventually becomes unwieldy, difficult to develop and support and ultimately becomes vulnerable by its dependence on certain key individuals. One solution to this is to simulate complex systems by integrating multiple, smaller models that collectively simulate the larger problem in question. That is, to build an integrated composition of previously independent numerical models and run them together allowing them to influence each other as they proceed through their respective time steps. The authors consider that any such solution should meet four key requirements:

(i) Allowing two-way exchange of results between the independent models in the composition as they proceed through their formulation.
(ii) Each component remaining sufficiently independent so that experts can remain in their disciplines, yet are able to communicate model outputs clearly where necessary at the interfaces between their coupled models.
(iii) Allowing the model interoperability to be undertaken flexibly and in a standardised fashion.
(iv) Enabling easy extensibility of the integrated composition to incorporate new parameters and to exchange similar numerical engines where appropriate.

OpenMI and FluidEarth have been undertaken to meet these challenges.

Background

Developed through considerable cooperation and joint working from leading hydraulic centres across Europe and part funded by the European Commission, OpenMI is a software component interface for the computational core (the engine) of a numerical model (Gregersen *et al.* 2007). Model engines are designed or modified to be 'OpenMI Compliant', thus enabling their inclusion in OpenMI integrated compositions. Previous versions of OpenMI have been used for many purposes including:

- river basin management (Makropoulos *et al.* 2010; Safiolea *et al.* 2011);
- dike seepage under transient boundary conditions (Becker & Schüttrumpf 2011);
- integrating agriculture, groundwater and economic models (Bulatewicz *et al.* 2010);
- water quality modelling (Shrestha *et al.* 2012);
- real time control of hydraulic structures (Becker *et al.* 2012); and
- beach plan-shape modelling (Sutherland *et al.* 2013).

As a response to European Union (EU) Water Framework Directive calls for integrated water management,

OpenMI itself was originally developed as a means for coupling existing models which would typically consider the interactions of environmental processes, in particular involving water (Moore *et al*. 2010). It has since been realised to be considerably more flexible, evolving into activities such as decomposing large models into smaller model components. It is now considered an interface standard between software components which can be applied to linking any combination of models, databases and associated tools (Lu & Piasecki 2012; OpenMI Association website 2012a).

OpenMI has been designed to allow two-way exchange of data between compliant components as they run, as explored by Elag & Goodall (2011). This would typically occur between two simultaneously running, timestepping model components which would send and/or receive data at specific timesteps as they proceed through their respective time intervals. In this way, the two model components can both influence the results produced by the other. The linked components may run asynchronously with respect to these timesteps or proceed through together. OpenMI also supports one-way passing of data from a driving component to a second, set up only to receive.

As an upgrade from the previous version 1.4, OpenMI 2.0 was released in December 2010 at a specially convened reception during an EU-US summit in Washington, DC (OpenMI Website 2012b). It incorporated a set of new features, some to build on the base from version 1.4 and others to replace or enhance the standard. These included the following:

- **Base Interfaces and Extensions** – A set of minimum 'base interfaces' for compliance, plus the addition of extensions (including an extension covering time and space dependent components). The essential OpenMI component is no longer forced to be time and space dependent, making the standard considerably more flexible and extensible. This allows different types of components to be incorporated e.g. those which vary in time and not in space; those which vary in space, but not in time or those which vary in both time and space (OpenMI Association Website 2010c).
- **Adaptors** – Taking over from the role of 'Data Operations' in OpenMI 1.4, 'Adapted Outputs' allow multiple, distinct adaptations, separate from the components themselves and the link, to take place. Again, this makes the standard more flexible and allows outputs and adapted outputs to be re-used by more than one OpenMI component (OpenMI Association Website 2010c).

IMPLEMENTATION

HR Wallingford's FluidEarth 2 is an implementation of OpenMI 2.0 consisting of a set of tools which provide an environment for the standard to be used. It uses the C# reference classes (OpenMI SourceForge project 2010). The tools are Open Source (FluidEarth SourceForge project 2012) and FluidEarth 2 also comes with a training website and examples, both ready for use and to act as templates for the user's own components. FluidEarth began as an implementation of OpenMI 1.4 and has been upgraded to FluidEarth 2 to meet the specification of this new OpenMI standard. It meets the OA (OpenMI Association) pledge to accompany each release of OpenMI with two tools:

- An SDK allowing model *developers* to easily make their model engines OpenMI compliant. FluidEarth 2 includes such an SDK to cover this requirement as a follow-up to that provided under OpenMI 1.4.
- A GUI allowing model *users* to build and run compositions of OpenMI compliant components. The FluidEarth 2 GUI, 'Pipistrelle', is a follow-up to the OpenMI 1.4 version of Pipistrelle and the OpenMI 1.4 Configuration Editor.

In addition to these and from its inception, FluidEarth incorporates three other aspects:

- A community of model providers and users.
- A set of websites/repositories: the FluidEarth portal at http://fluidearth.net including document libraries, news, discussion, case studies and community contact details; a model catalogue allowing the listing of model engines and their instances at http://catalogue.fluidearth.net; a source forge repository with open source application code at http://sourceforge.net/projects/fluidearth/ and an extensive training website at http://training.fluidearth.net.
- A library of models available for compositions.

FluidEarth has been in operation since 2008, originally supporting OpenMI version 1.4. OpenMI 2.0 is the current

supported standard, although compatibility with version 1.4 has been maintained. The tools and training for OpenMI 1.4 are still available from the FluidEarth portal and Pipistrelle includes a facility to allow OpenMI 1.4 components to be automatically made OpenMI 2.0 compatible.

The philosophy behind the FluidEarth 2 SDK and Pipistrelle follows that of OpenMI version 2.0 itself: openness and flexibility. OpenMI 2.0 includes a base standard and extensions; Pipistrelle gives rich base functionality and also plug-ins. The tools themselves are designed with a high-degree of usability and are supported by clear and simple training, leading the user into the principles of use of OpenMI 2.0 with FluidEarth 2 with highly straightforward use cases designed to demonstrate the capabilities of the toolset. Templates are provided with detailed explanations allowing users to adapt these simple examples into real use cases.

Compositions can be set-up and loaded into Pipistrelle using the menus, although it is also possible to configure a composition by editing the configuration files (xml) directly and running through the console. The tool runs the compositions using a 'pull' approach. The most common use case involves timestepping components which proceed through time in a series of intervals calculating values at each. One of the components is assigned a 'trigger' which controls the composition. Components then demand data from one another, waiting while the requested component runs timesteps until it can meet the request. The 'adaptors' concept from OpenMI version 2.0 is implemented directly through an additional set of menus, allowing connections between components to apply functions to the data passed along the linkages. These adaptors are independent of both components; a departure from the OpenMI 1.4 concept where this functionality resided within one or other of the components linked.

With usability and interoperability seen as specific goals of the FluidEarth 2 implementation, three features have been built into Pipistrelle and the FluidEarth SDK since its original release in 2012. The first of these is the ability to save a FluidEarth OpenMI 1.4 composition (containing components prepared under the previous version of FluidEarth using OpenMI 1.4) as a FluidEarth 2 composition. This facility is supplied with downloads of Pipistrelle from July 2013 (it existed in previous versions but did not stand up to full testing) and works as long as the 'data operations' aspect of the standard has not been used in these components. Figure 1 gives a screenshot of this facility in Pipistrelle.

The second feature built into versions of Pipistrelle available from the summer of 2013 is the ability to view spatial datasets. This facility is added as a plug in and gives the user a two-dimensional view of the node sets related to the models in the composition. This can be particularly

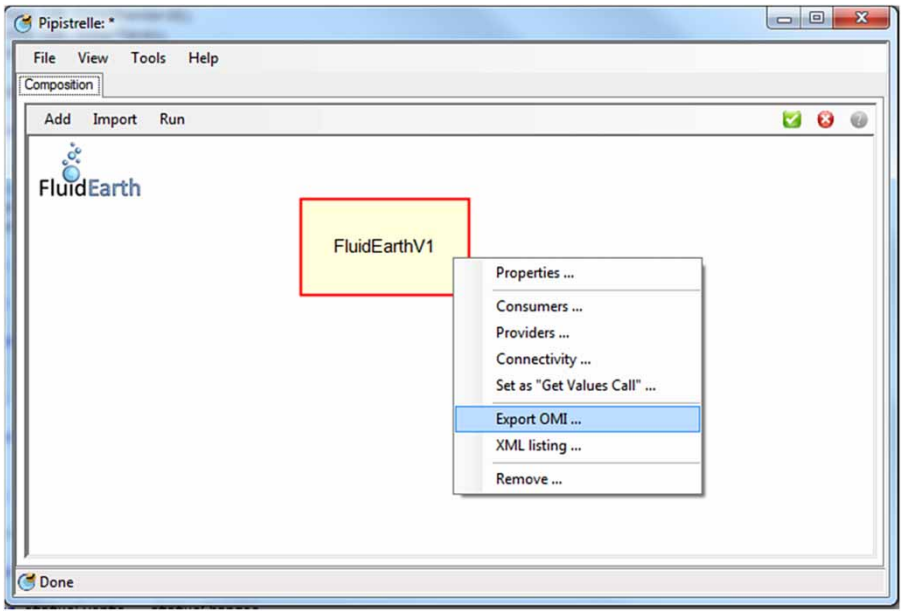

Figure 1 | Using Pipistrelle to save an OpenMI 1.4 component as an OpenMI 2.0 component.

important when different meshes (or even different spatial structures) are required to be connected. Nodes that are required to pass data between one another can be more easily identified and adaptations followed. Figure 2 shows a screenshot of the 'Spatial View' plug-in which is built around the open source DotSpatial geographic information system library (http://dotspatial.codeplex.com/) giving base Geographic Information System (GIS) functions to the user. It shows spatial layers being loaded and viewed.

The third feature is called 'Component Builder', again available from summer 2013. This facility is designed to avoid any FORTRAN programmers having to write code in C#. The C# shell code is built automatically by component builder, based on certain knowledge provided about the FORTRAN module(s) to be wrapped. Figure 3 shows a screenshot of the component builder plug-in at the point in the process where the user is selecting a spatial definition to match that of a FORTRAN model which is undergoing the wrapping process to form an OpenMI 2.0 component.

Native language development

The FluidEarth implementation of OpenMI version 2.0 has been achieved through the use of two development languages: C# and FORTRAN. Visual Basic (.net) is expected to be a corollary of this approach, but the perceived lack of potential components to be implemented in Visual Basic resulted in full and complete testing of Visual Basic counterparts of all the components offered in C# to be omitted in favour of some simple verifications. This can be completed at a later date should demand for components written in Visual Basic be demonstrated. Components written in other languages such as Python and C++ are feasible, but this has not been tested to date.

At run time the FluidEarth Pipistrelle GUI connects components in a composition using .net Framework connectivity to load and execute objects and their methods. Components which are developed in a native compiled language (that is code that is unmanaged by the Common Language Runtime) such as FORTRAN or C, are wrapped in a thin C# (or other .net language) wrapper class which performs the work of communicating with the native code itself. The native code base must implement a defined set of functions (provided by a template) to allow the managed code wrapper to control the running of the native code. These are given in Table 1. More detailed descriptions are available in the FluidEarth Help documentation (FluidEarth SourceForge Project 2012).

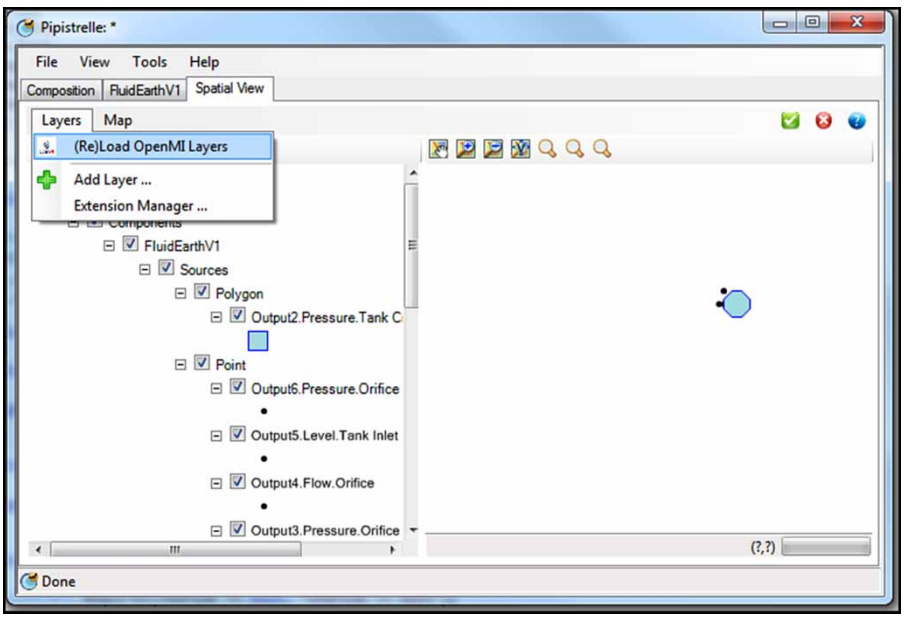

Figure 2 | Pipistrelle Spatial View plug-in screenshot.

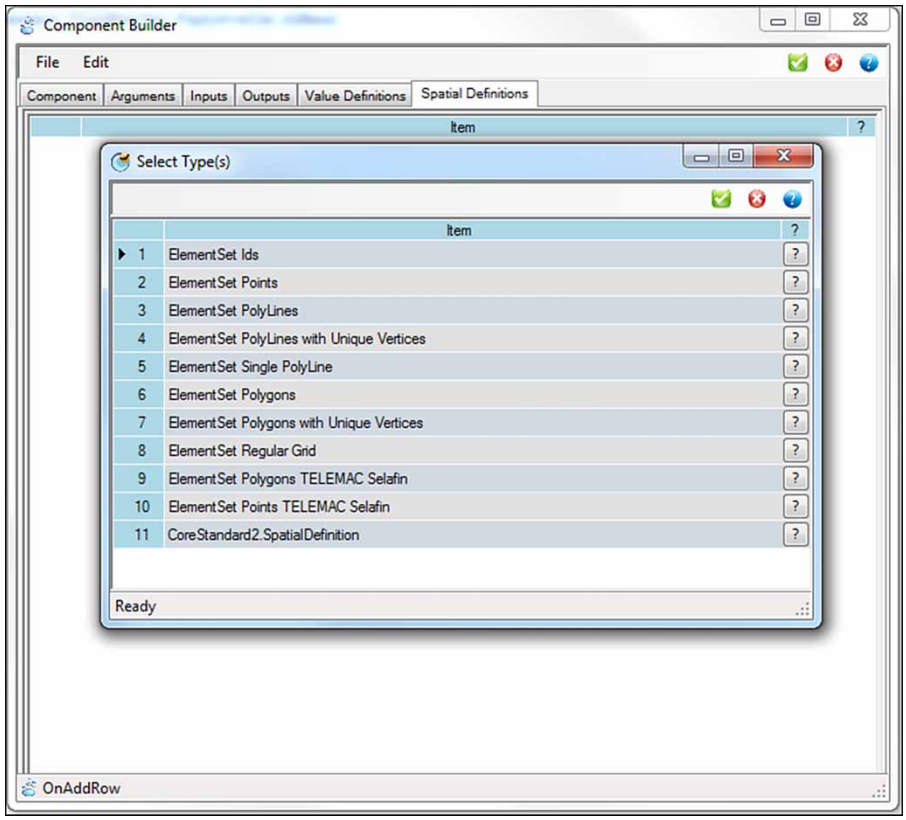

Figure 3 | The Pipistrelle Component Builder plug-in.

Native code is unmanaged. It is not controlled by the Common Language Runtime of .net nor the Java Virtual Machine environment; its memory allocation is handled directly; run time type checking and reference checking are also uncontrolled by .net/Java. As the native code is unmanaged it is essential it is either written as thread-safe (that is, it runs in a manner which guarantees that other executing threads will not be destructively interfered with) or, ideally, run in a completely separate process via the FluidEarth remoting protocols.

Remoting is a .net technology for handling communication between objects across computer and network boundaries. If a component in a composition is to be run outside of the managed Pipistrelle application process (because it is not guaranteed to be thread-safe like a native code engine) then the remoting option of the component will need to be set to something other than 'InProcess'. Once selected a communications technology can be selected from one of IPC (Inter-process communication), AutoIPC (Inter-process communication where the identifiers are managed automatically), TCP (Transmission Control Protocol) and HTTP (Hyper Text Transfer Protocol). This can be configured per component in the Pipistrelle GUI which offers the options for executing a component in Table 2. The approach used does not exclude future development of other protocols, methods of communication and runtime control.

EXAMPLE CASES

The objective sought by the FluidEarth 2 toolkit (SDK and GUI) is to easily enable models (and other valid components) to be made OpenMI compliant and to provide a user-friendly graphical interface to allow users to assemble and run compositions. A set of examples was put together to progressively test the functionality of the toolkit, from a low level of complexity to a level expected by a typical 'real world' example of hydraulic modelling. This technical progression and the

Table 1 | Template functions for native code to implement; Mandatory (*M*), Optional (*O*)

	Function/Subroutine signature
M	function FLUIDEARTH2_ENGINE_PING() Used to establish that the engine has been successfully instantiated.
M	function FLUIDEARTH2_ENGINE_SUCCESSMESSAGE(success_code, message) Returns a message determined by the success_code parameter
M	subroutine FLUIDEARTH2_ENGINE_INITIALISE(args, success_code) Initialises FluidEarth2.Sdk.BaseEngine: Arguments from the supplied text XML and then uses argument helper functions to set specific engine parameters.
M	subroutine FLUIDEARTH2_ENGINE_SETARGUMENT(k, v, success_code) Initialises the argument specified by k with the value v returning the result of the operation in success_code.
M	subroutine FLUIDEARTH2_ENGINE_SETINPUT(engine_variable, element_count, element_value_count, vector_count, success_code) Called for each active IBaseInput and informs this code that the engine variable specified by engine_variable has been activated for this run the so engine must use these values when they arrive; element_count is the number of elements in the corresponding element set; element_value_count is the number of values per element; vector_count is the size of the vector for each value.
M	subroutine FLUIDEARTH2_ENGINE_SETOUTPUT(engine_variable, element_count, element_value_count, vector_count, success_code) Called for each active IBaseOutput and informs the code that the engine variable specified by engine_variable has been activated for this run so engine must use these values when they arrive; element_count is the number of elements in the corresponding element set; element_value_count is the number of values per element; vector_count is the size of the vector for each value.
M	subroutine FLUIDEARTH2_ENGINE_PREPARE(success_code) The place to dynamically allocate memory for arrays and other requirements.
O	subroutine FLUIDEARTH2_ENGINE_SETINT32S(engine_variable, missing_value, values_size, values, success_code) Called for each active IBaseInput before a call to Update(); this is used to set the values of a 32 bit integer type variable prior to the Update();
O	subroutine FLUIDEARTH2_ENGINE_SETDOUBLES(engine_variable, missing_value, values_size, values, success_code) Called for each active IBaseInput before a call to Update(); this is used to set the values of a double precision type variable prior to the Update();
O	subroutine FLUIDEARTH2_ENGINE_SETBOOLS(engine_variable, missing_value, values_size, values, success_code) Called for each active IBaseInput before a call to Update(); this is used to set the values of a Boolean type variable prior to the Update();
M	subroutine FLUIDEARTH2_ENGINE_UPDATE(success_code) Called to perform a calculation and modify the component's time step.
O	subroutine FLUIDEARTH2_ENGINE_GETINT32S(engine_variable, missing_value, values_size, values, success_code) Called for each active IBaseOutput after a call to Update(); this is used to retrieve the values of a 32 bit integer variable immediately after a call to Update().
O	subroutine FLUIDEARTH2_ENGINE_GETDOUBLES(engine_variable, missing_value, values_size, values, success_code) Called for each active IBaseOutput after a call to Update(); this is used to retrieve the values of a double precision variable immediately after a call to Update().
O	subroutine FLUIDEARTH2_ENGINE_GETBOOLS(engine_variable,missing_value,values_size, values,success_code) Called for each active IBaseOutput after a call to Update(); this is used to retrieve the values of a Boolean variable immediately after a call to Update().
M	subroutine FLUIDEARTH2_ENGINE_FINISH(success_code) Called to dispose of any dynamically allocated resources – as may have been allocated in Prepare().
M	function FLUIDEARTH2_ENGINE_GETCURRENTTIME(success_code) Called to return a double precision number representing the current time as known to the component.

testing examples used has been built into a training website (Cleverley 2012), giving a comprehensive introduction to FluidEarth and its underlying concepts. First, a simple, stand-alone model is constructed. It has no geospatial attributes, offers a single parameter as output and receives a single identical parameter as input. Using just this component, it is then possible to construct the very simplest compositions: a single OpenMI 2.0 component running

Table 2 | FluidEarth 2 component execution options

Component remoting option	Description
In Process	Runs the component 'in process' - the usual mechanism for .net managed components. Each instance of a given managed component will have its own memory space but each instance of unmanaged components will share a single memory space so the danger is that each instance of a given unmanaged component might overwrite data from another instance of the same unmanaged component.
IPC Auto	Runs the component in a separate process using inter-process communication protocols whilst automatically assigning port and object identifiers – this will ensure that, in a given composition, instances of the same unmanaged component will run with its own memory space.
IPC	Runs like IPC Auto but requires the user to specify the object and port identifiers (not used in Cleverley (2012)).
TCP	Runs the component via TCP protocol (not used in Cleverley (2012)).
HTTP	Runs the component via the HTTP protocol (not used in Cleverley (2012)).

alone and the natural corollary of another simple composition linking two identical pond instances together – one pond drains into its twin. Further examples then add an adaptor between the two components (in this case to perform a unit translation where one pond drains in centilitres into another which requires input in millilitres), geospatial structures (linear boundaries) and geospatial interpolation (between these linear boundaries, but with differing numbers of nodes). This series of examples is performed using models and adaptors written in both C# and FORTRAN.

When defining suitable examples to demonstrate the development of Open MI 2.0 compliant FluidEarth components the FluidEarth 'engine pattern' was used. This results in a component interface class which is separate from its related engine class. An interface class is a 'class that primarily defines a protocol, but does not provide an implementation. This means they only describe the expected behaviour …' (Google Web Definitions 2013). This separation allows changes to the engine operation whilst maintaining the interface exposed to the Pipistrelle GUI. The engine class itself is implemented using one of the interfaces given in the FluidEarth 2 SDK *FluidEarth2.Sdk.Interfaces* – either *IEngine* or *IEngineTime*.

The term 'Engine' refers to the executable code that the user wishes to 'wrap' or develop to make it 'OpenMI compliant'. The Engine Pattern simply means Engine code that can be used via the interfaces FluidEarth2.Sdk.Interfaces.IEngine or FluidEarth2.Sdk.Interfaces.IEngineTime. OpenMI does not require the use of the Engine Pattern to make code compliant. It is a simplification which, if possible, then allows SDK providers to provide libraries of code to simplify the compliancy task. Hence, if a user's code can be reformulated to use one of these engine interfaces it can be easily implemented using the FluidEarth2_SDK.

Time stepping components, that is models or data providers which offer data over a timeline divided into timesteps each of which holds data values, have been used throughout since most current requirements fall into this category. In this case, the results of the engine change over time and the component exposes a time 'horizon' as an argument. Hence the selected component uses the *FluidEarth2.Sdk.Interfaces.IEngineTime* interface.

The simple pond

This Simple Pond example, taken from the training material (Cleverley 2012), illustrates a simple form of OpenMI component. It allows the user to grasp some basic aspects of using OpenMI with FluidEarth 2 as well as offering some template code. In addition to the necessary default OpenMI required arguments, this component (a Pond) has the arguments **capacity**, **currentlevel** and **flow**. These are denoted in Figure 4, below.

flow indicates the amount of water which overflows out of the component each time step; **currentlevel** gives the current water level of the Pond at any given time (at the

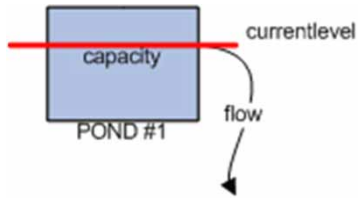

Figure 4 | Simple Pond OpenMI component schematic view.

beginning, during the run and at the end of the run); **capacity** gives the amount of water contained in the component which must be exceeded before any overflows. In addition, the component has one input: **inFlow** and one output: **outFlow**, both variables whose value may change at run time. The base functionality of the FluidEarth SDK and Pipistrelle has been demonstrated through the most simple composition with this model, stand alone, in both C# and FORTRAN. In addition, another simple composition is possible by connecting two identical copies of this pond, connecting the inFlow of the first pond into the outFlow of the second.

Adaptors are required when the output of one model cannot be directly connected to the input of another. To demonstrate the basic adaptor definition and function in FluidEarth an adaptor was developed to perform a simple unit transformation against a single input, multiplying it by 10 and giving a single output. This generic function is used in this case to convert centilitres to millilitres. In this way the user building the composition can focus on the details of building an adaptor rather than the function of the adaptation itself.

The next composition, depicted in Figure 5 and again taken from the training material (Cleverley 2012), gives an instance of the Pond component (Pond #1) overflowing at a certain rate when its capacity is exceeded. The adaptor modifies its input from centilitres to millilitres and passes that value onto the second component (Pond #2) as the inFlow input. Pond #2 fills up until it too begins to overflow.

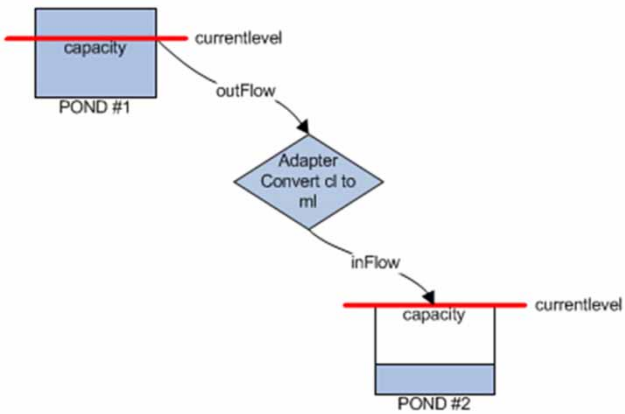

Figure 5 | FluidEarth 2 composition involving a simple adaptation.

The two-dimensional pond

In moving towards a more typical modelling solution, the pond theme is continued, but a geospatial structure is added to the components. The two-dimensional pond example is designed to begin to illustrate usage of spatial structures and provide template code for users. It is also taken from the training material (Cleverley 2012) and presents a straightforward use case. Pond II offers output across arrays at each boundary, evenly spread across each length to represent water transfer across the entire length of each pond edge (see Figure 6).

When two such pond components are joined in a composition the action is similar – fluid will flow from one part of the pond to another as it drains into a second, identical pond component along a boundary. The nodes of the eastern boundary of the first pond match to the nodes on the western boundary of the second pond one-to-one, with values passed directly between the two.

In removing the restriction of the connected boundaries being of the same array dimension an adaptor is required to interpolate between boundaries of different sizes. In Figure 7 the 'ten-node' eastern boundary of Pond #1 needs to be connected to the 'five-node' western boundary of Pond #2.

Without this one-to-one mapping of outputs to inputs, the 2d pond adaptor provides an interpolation to allow values to be passed between components. Of course, any conversion between these two node-sets is possible, the

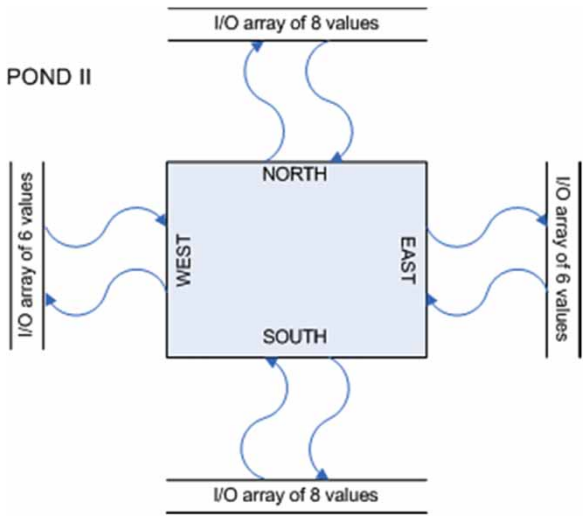

Figure 6 | The two-dimensional pond component with water transfer across pond edges.

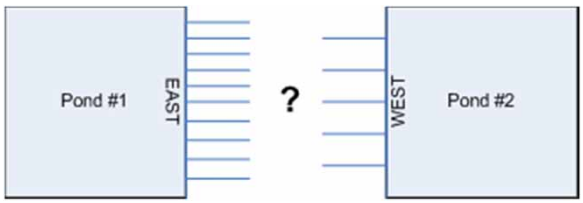

Figure 7 | Connecting ponds of different boundary dimensions.

adaptor concept is given as a placeholder for the implementation of any chosen algorithm.

In Cleverley (2012), simple examples have been chosen, since the emphasis is on learning how to build and incorporate an adaptor and not on working with complex data aggregation algorithms. This allows the student to learn how to build an adaptor which handles two-dimensional data rather than showing the behaviour of a more realistic example. A two-dimensional composition connects the East boundary Pond #1 to the West boundary of Pond #2 using a two-dimensional aware adaptor to interpolate the values where necessary.

Coupling two timestepping models with a simple dam-break test case

A more involved coupling scenario is now considered. The composition comprises two timestepping models coupled together in Pipistrelle via an adapter. The model OTT2D is a 2DH (2 dimensional horizontal) NLSW (non-linear shallow water) solver. The OTT2D solver employs a collocated (cell-centred) finite volume scheme. A detailed description of the OTT2D solver is beyond the scope of this paper and, as such, a full description of the OTT2D model can be found in Hubbard & Dodd (2002). The Exner solver solves the sediment continuity equation employing the simple, first-order accurate, node based 'upstream' finite-difference scheme of Perdreau & Cunge (1971). The sediment continuity (Exner) equation relates bathymetric evolution to sediment flux divergence via a sediment transport formula and can be written in vector notation according to Equation (1),

$$\frac{\partial B}{\partial t} = -\nabla \cdot \vec{q} \tag{1}$$

where $B = B(x, y, t)$ is the bed height relative to a datum level and $\vec{q} = \vec{q}(\vec{u}, h)$ is the vector of sediment fluxes.

These models are coupled together in order to allow a subsection of the OTT2D model bathymetry to evolve based on the hydrodynamic conditions. The OTT2D mesh is a different spatial representation than that of Exner, in fact, in the example below the Exner mesh is nested within the OTT2D mesh (see Figure 8). An adaptor is therefore required to allow the models to pass data. A simple spatial, BIA (Bilinear Interpolation Adaptor) is used. It is effectively a linear interpolation based on a triangulation of the input point set. The point set comprising the OTT2D mesh is first Delaunay triangulated and the bounding triangle (i.e. the triangle that encloses the interpolation point) for each interpolation point on the Exner sub-mesh is identified. A weighted average based on splitting the bounding triangle into three sub-triangles with the interpolation point as a common vertex is used. Areas of the bounding triangle and three sub-triangles are computed using the general formula given by Braden (1986). Values at each vertex of the original bounding triangle are then weighted according to the relative weights of each the three-sub triangles to the bounding triangle to give the value at the interpolation point. The two models that comprise the composition, OTT2D and Exner, are run on distinct meshes; the OTT2D mesh is cell centred whilst the Exner mesh is node centred so the meshes are not coincident. The Exner mesh comprises a sub-domain of the larger OTT2D mesh as illustrated in Figure 8, which clearly shows the node centred finite difference Exner mesh (depicted in white) nested within the cell centred finite volume mesh of OTT2D (depicted in grey), with the inset section illustrating that

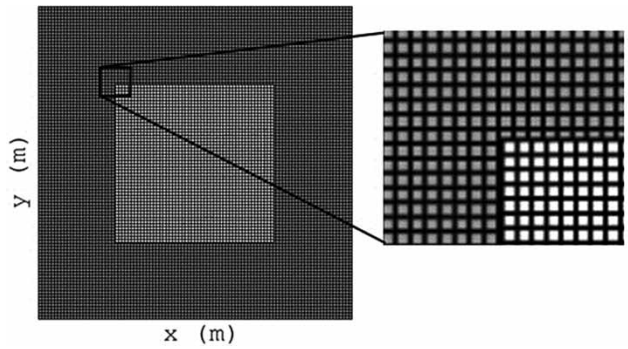

Figure 8 | The experimented configuration with the Exner Mesh within the OTT2D mesh.

the meshes are not coincident. The OTT2D solver solves the 2DH NLSW equations to give the time evolution of the water depth, h, and depth-averaged velocity components u and v. Velocities output from OTT2D are interpolated at each point on the Exner mesh using the adapter described above and the associated sediment fluxes are computed according to the Grass (1981) formula in Equation (2),

$$\vec{q} = A(u|\vec{u}|^2, v|\vec{u}|^2) \tag{2}$$

where A is a dimensional transport constant (dimensions $s^2\,m^{-1}$) whose value can be related to sediment density and grain size (d_{50}) (see e.g. Hudson 2001).

The Grass formula is used to close the Exner equation in this illustrative example as it is the simplest total load sediment transport formula available. Test specific details on the mesh dimensions are provided in the test case section.

A test case is now considered which moves towards that typical of 'real world' usage of these models. It consists of a simple wet-wet dam-break in a closed box and is used purely to illustrate the coupling of two-different timestepping models via an adaptor. The initial conditions for the simulation are shown in Figure 9 and comprise still water of depth 1 m with a 1 m high 20 m × 20 m block, or reservoir, of water centred at $x = 55$ m, $y = 55$ m. The OTT2D mesh has a uniform mesh spacing of $\Delta x = 1$ m and $\Delta y = 1$ m.

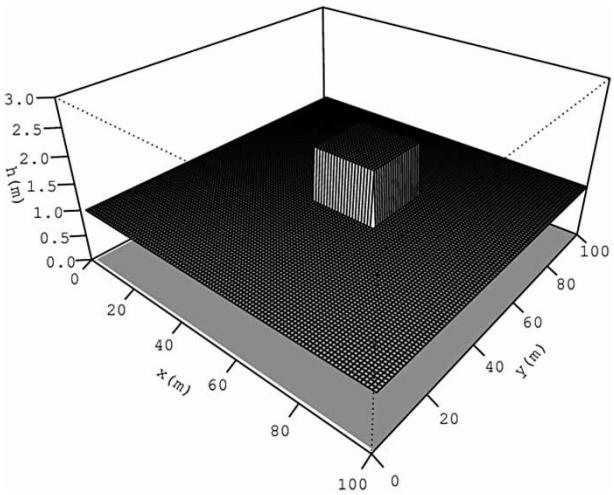

Figure 9 | Initial dam-break test case conditions.

The Exner mesh has its origin at $x = 25.25$ m, $y = 25.25$ m and also uses a uniform mesh spacing of $\Delta x = 1$ m and $\Delta y = 1$ m (see Figure 9). All of the water is initially at rest and at time $t = 0$ the dam 'walls' are assumed to vanish instantaneously creating a shock wave, or bore, that propagates outwards towards the domain boundaries.

We note here that the coupling is one-way with OTT2D passing data to the Exner solver; the water movement deforms the bathymetry but changes in the bathymetry are not fed back to OTT2D. This limitation has been applied due to modelling complexities with this example: when running a model that updates the bed at a different timestep to the flow, instabilities are difficult to avoid and water depth must be corrected to account for bed change. Addressing these issues is beyond the scope of this simple example. An output of the Exner solver is the total bed evolution since the beginning of the simulation $E(i, j)$ which is computed according to Equation (3),

$$E(i,j) = \int_0^T \Delta B(i,j)\, dt \;\forall\; i,j \tag{3}$$

where T is the frame time for the simulation, and i, j are the x and y indices, respectively, of the finite difference mesh employed by the Exner solver.

Figures 10–12 show the results of the simulation paused after 4 s of simulation time; these include the water surface, velocity vectors and bed evolution.

Figure 11 shows the outputs of OTT2D as velocity vectors which are passed to Exner through the BIA adaptor at the same timestep.

Time variant two-way data exchange

We now consider a two-way exchange of data between two OpenMI components in a single composition as given in the training material (Cleverley 2012). This common requirement of OpenMI compositions allows two models to influence each other as they run. Components pass data to each

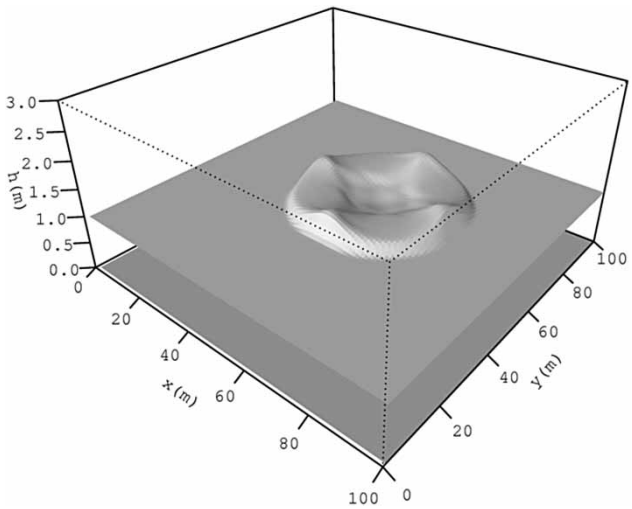

Figure 10 | A snapshot of the water surface and bathymetry (bottom) at $t = 4$ s for the dam-break in a box test problem.

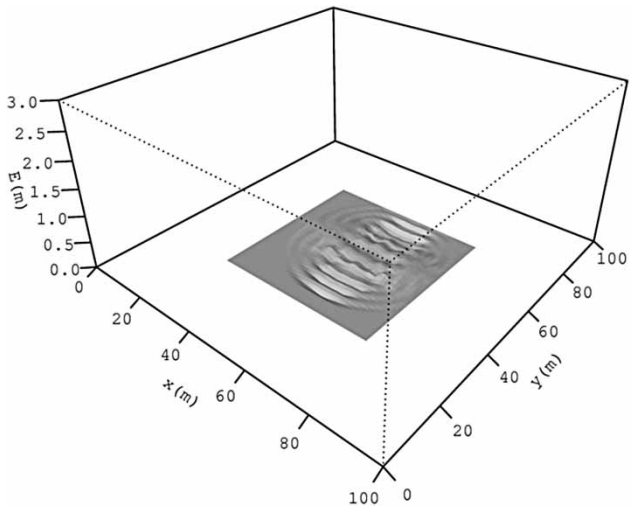

Figure 12 | The total bed evolution at $t = 4$ s computed in the subdomain of the main mesh that is used by the Exner solver.

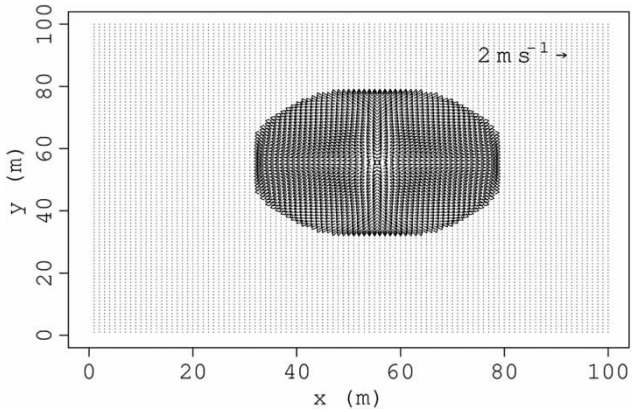

Figure 11 | Velocity vectors at $t = 4$ s as output from the OTT2D solver to the BIA adapter.

other on demand as the composition runs, with each model advancing its internal time. Component A requests data from Component B which runs through sufficient internal timesteps until it can fulfil Component A's request. Similarly Component B may reach a point where it needs to request data from Component A. Component A then runs through sufficient timesteps until it, in turn, can fulfil Component B's request. One component will be the prime driver of this composition (connected to the run trigger) and its completion will signal the completion of the composition itself.

Such a bi-directional exchange of data between components may result in deadlock: Component A is waiting for Component B to fulfil its request for data, but Component B cannot do so until it receives data from Component A. Neither component can proceed and the composition fails to complete successfully. Pipistrelle provides a solution to prevent such deadlock situations occurring: if a component is asked for information that it cannot provide by computation (for example because it would be relying on data supplied from the requesting component) then the component is forced to provide a value, even if it has to approximate.

Morita & Yen (2002) is an example of a model coupling where the value for the previous timestep is used. Pipistrelle, however, is also designed to cover situations where the timestep values of the two models may differ considerably. If the default within Pipistrelle is to supply the previously computed value and a coarse timestep is being used in one component and a finer timestep in another component, then the supplied value may be considerably out-of-date. Accordingly, Pipistrelle uses an extrapolation from previously computed values – a polynomial interpolation based on a defined number of previously calculated results. The request-reply mechanism in use is described in the OpenMI Standard 2 Specification document (OpenMI Association Website 2010d).

By way of example, consider two reservoirs of liquid, A and B. They are connected to each other by two independent channels. One channel only allows water to be pumped from reservoir A to reservoir B and the other channel allows only the reverse, from B to A.

We wish to apply the following rules to the system:

- At a given time, if reservoir A contains more liquid than B then A will pump a certain quantity of liquid (QAB) to B. The quantity pumped (QAB) will be calculated so that, when added to the current level of B, it will not exceed the capacity of B, nor exceed an arbitrary maximum value, nor allow the level of A to drop below zero.
- Equally, if B contains more liquid than A then B will pump a quantity of liquid (QBA) to A. Again, this amount will be calculated so that the level of A won't then exceed its capacity, nor an arbitrary maximum value, nor allow the level of B to drop below zero.

As such:

- No component should ever be allowed to overflow.
- No component should ever pump more than the minimum of its current level and an arbitrary maximum.

Each component will require variables representing its current level and the amount it can pump. Each component will also require an 'input exchange item' representing the quantity of water received and an 'output exchange item' representing the quantity of water pumped. In the composition quantity of water received (A) is linked to quantity of water pumped (B) and quantity of water received (B) is linked to quantity of water pumped (A). So the composition is set up to pass water both ways between the components. Furthermore, as each component will need information from the other component before calculating the amount it is about to pump, each component will require 'output exchange items' exposing its own current level and capacity, and input exchange items indicating the current level of the other component and its capacity.

As illustrated in Figure 13, each component will comprise arguments (Capacity, Level and QuantityToPump), inputs (QuantityReceived, OtherComponentLevel and OtherComponentCapacity) and outputs (QuantityPumped, Level and Capacity).

```
component    =    arguments    =    Capacity
                                +    Level
                                +    QuantityToPump

             +    inputs       =    QuantityReceived
                                +    OtherComponentLevel
                                +    OtherComponentCapacity

             +    outputs      =    QuantityPumped
                                +    Level
                                +    Capacity
```

Figure 13 | Reservoir OpenMI Component arguments, inputs and outputs.

We follow the pull driven approach favoured by the FluidEarth Pipistrelle GUI and so reservoir B will request liquid from reservoir A and reservoir A will request liquid from reservoir B. Each request for data will advance the time step for each component. Figure 14 shows a screenshot of Pipistrelle with this composition loaded.

The initial conditions represent the composition in an unbalanced state. Reservoir A begins with a level of 59 and B with 40. B then requests liquid from A and in this manner the composition progresses from the unbalanced state, where A has more liquid than B, to a state with a degree of equilibrium as seen in Table 3. Units are arbitrary in this notional example but consistent across the composition. Note, also, that it takes time for water to proceed from one reservoir to another.

Figure 15 gives the water levels of both reservoirs as the composition runs. Figure 16 shows the water pumped from each reservoir. Instability occurs at the equilibrium point causing the composition to oscillate and neither reservoir is able to find a stable level.

Time invariant two-way data exchange

The time variant two-way data exchange described above represents a typical two-way OpenMI 2.0 composition; one for which Pipistrelle was originally conceived and has been designed to address. However, it is also possible for two components to require exchanging data with each

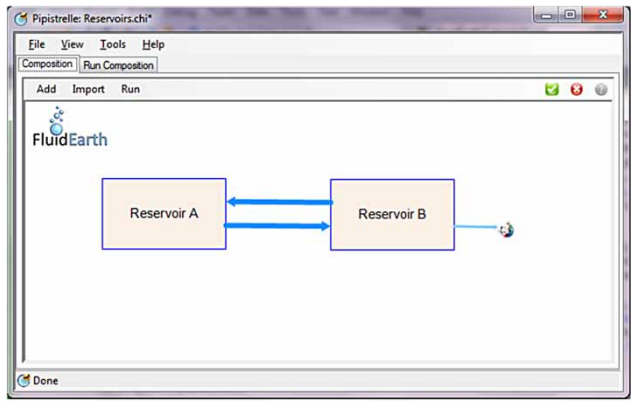

Figure 14 | A basic two-way exchange composition.

Table 3 | Two-way reservoir composition results

	Reservoir A			Reservoir B		
Current Time	**Level**	**Received**	**Pumped**	**Level**	**Received**	**Pumped**
2013-05-31 23:00:00Z	59	0	1	40	0	0
2013-05-31 23:05:00Z	58	0	1	41	1	0
2013-05-31 23:10:00Z	57	0	1	42	1	0
2013-05-31 23:15:00Z	56	0	1	43	1	0
2013-05-31 23:20:00Z	55	0	1	44	1	0
2013-05-31 23:25:00Z	54	0	1	45	1	0
2013-05-31 23:30:00Z	53	0	1	46	1	0
2013-05-31 23:35:00Z	52	0	1	47	1	0
2013-05-31 23:40:00Z	51	0	1	48	1	0
2013-05-31 23:45:00Z	50	0	1	49	1	0
2013-05-31 23:50:00Z	49	0	1	50	1	0
2013-05-31 23:55:00Z	49	0	0	50	1	1
2013-06-01 00:00:00Z	50	1	0	49	0	1
2013-06-01 00:05:00Z	50	1	1	49	0	0
2013-06-01 00:10:00Z	49	0	1	50	1	0
2013-06-01 00:15:00Z	49	0	0	50	1	1
2013-06-01 00:20:00Z	50	1	0	49	0	1
2013-06-01 00:25:00Z	50	1	1	49	0	0
2013-06-01 00:30:00Z	49	0	1	50	1	0
2013-06-01 00:35:00Z	49	0	0	50	1	1
2013-06-01 00:40:00Z	50	1	0	49	0	1
2013-06-01 00:45:00Z	50	1	1	49	0	0
2013-06-01 00:50:00Z	49	0	1	50	1	0
2013-06-01 00:55:00Z	49	0	0	50	1	1
2013-06-01 01:00:00Z	50	1	0	49	0	1
2013-06-01 01:05:00Z	50	1	1	49	0	0

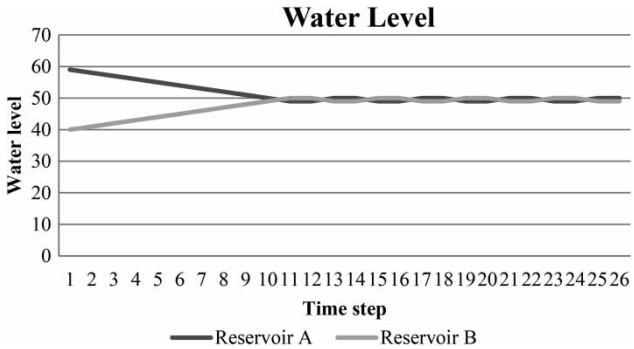

Figure 15 | Two-way reservoir composition water levels.

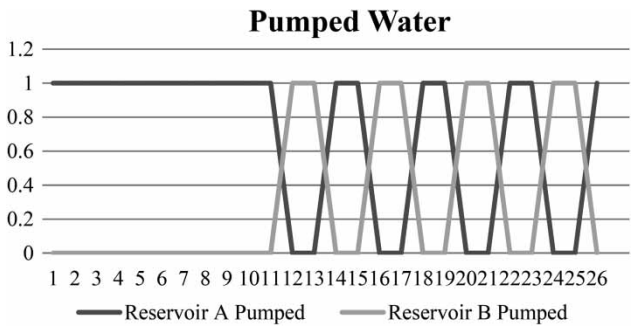

Figure 16 | Two-way reservoir composition pumped water.

other before completing an individual composition timestep. Such an example could be considered a 'time invariant' two-way data exchange. So the bi-directional exchange of information needs to take place within a single timestep. This could be required in compositions where components need to assess the status of other components without advancing their computation. It is possible that a component will wish to gather information about the state of other components before making decisions about its own computation. A time invariant version of the exchange items can be used to ensure that certain exchanges won't progress the clock of the component from which the information is requested.

Consider, as an example the simple FluidEarth Pond model, refactored as 'ConditionalPond'. ConditionalPond only allows water to flow from itself to a downstream component if the current level of the downstream component plus the flow it would receive does not exceed its own capacity. This condition must be determined in such a way that the downstream component's time remains unaffected. This requires developing a TimeInvariant set of ValueSetConverters which allow input and output items to be exchanged without influencing the component's time step.

We connect two independent instances of ConditionalPond together in a two-way connection so that ConditionalPond is type of both upstream and downstream components. Then the ConditionalPond class must implement the appropriate behaviour for the condition where it may be both up and downstream of other similar components. Hence it has both input and output exchange items corresponding to Capacity, CurrentLevel and Flow. Naming our upstream component 'Upstream' and our downstream component 'Downstream' we can represent the model composition as in Figure 17.

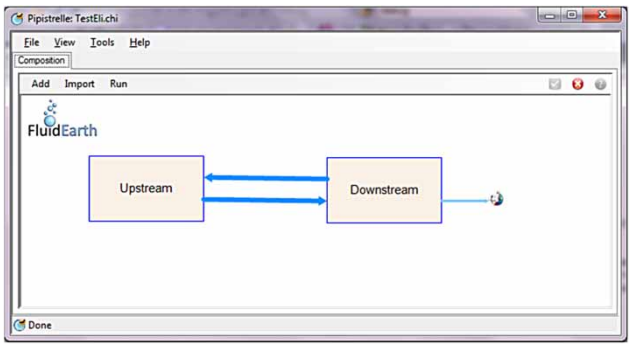

Figure 17 | Two-way connections using the same 'ConditionalPond' class.

The arrow pointing from Upstream to Downstream represents the exchange item Flow. The arrow pointing from Downstream to Upstream represents the exchange items CurrentLevel and Capacity.

When the composition begins, the trigger will request Flow from Downstream. Once Downstream has satisfied the request, its current time will increment. However, in order to satisfy the request from the trigger, Downstream will request Flow from Upstream. Critically, within the same time step, and in order to determine the value of Flow, Upstream will request CurrentLevel and Capacity from Downstream and will modify Flow so that CurrentLevel + Flow <= Capacity.

This request for CurrentLevel and Capacity will not result in the Downstream current time being incremented, hence the nomenclature 'TimeInvariant' applied to the relevant ValueSetConverter (see Figure 18). An IValueSetConverter is a FluidEarth implementation interface that factors out the runtime specific implementation details of a specific OpenMI IBaseExchangeItem interface. Typically this is where the logic for implementing the data transfer resides. Thus a TimeInvariant version results in no increment in the timestep for the component when the exchange item is updated.

This capability allows the building of compositions where component interchange can depend on the state of other components at a given time step and, effectively, makes the state of the components in a given composition available to all other components at run time. This method facilitates such approaches as agent-based modelling where a population of individual agents is modelled by

```
var converterInCurrentLevelNoTime = new
ValueSetConverterTimeInvariantEngineDouble(en_inCurrentLevelNoTime, 0.0, 1);
```

Figure 18 | TimeInvariant ValueSetConverter code snippet.

each agent being aware of the state of other agents at any given time. The bi-directional passing of data allows them to undertake tests of each other's attributes within timesteps before each makes any decisions about their next exchange.

SUMMARY AND OUTLOOK

FluidEarth 2 is an implementation of OpenMI version 2.0 which seeks openness, flexibility and usability. The successfully executed examples using the FluidEarth 2 SDK and Pipistrelle given above, range from simple one-way compositions to those more typical of real industry or academic requirements. These examples have been built in C# and FORTRAN with VB usage seen as a corollary. The model coupling process is improved and more accessible to less technical users. Using the Pipistrelle GUI, compositions can be built utilising compatible components from different suppliers in a high usability environment. The same GUI is used for executing the compositions offering the same desired level of usability. It has been necessary to apply a detailed level of instruction for building components using the SDK since this is the most involved procedure, tending to be the most esoteric. Usability is higher with the natural C# development language (that of Pipistrelle and the SDK) although FORTRAN compositions are also readily accessible.

The most involved of the compositions, that of coupling together OTT2D and Exner via a simple adapter, from a user's point of view yielded a generally positive experience, indeed a strong improvement from the FluidEarth implementation of OpenMI 1.4. The introduction of adapters as a concept has dramatically improved the usability of implementations of OpenMI 2.0 such as FluidEarth 2 for the linking together of two or more area-type models running on two or more distinct meshes. Responsibility for the adaptation now lies independently from the two model components. These can then remain the same for a variety of compositions with adaptors coded independently and applied as required. For a user who is familiar with the model(s) to be wrapped, the conversion of an existing model to a FluidEarth 2 component is a relatively straightforward task. Both models that were wrapped for this composition had FORTRAN as the native language and the FORTRAN template provided in the FluidEarth 2 download (Harper *et al.* 2012) greatly facilitated the wrapping procedure. It is noted that the simple adapter presented here cannot be expected to conserve quantities that should be conserved. To this end it precedes the generation of a library of adapters that are suitable for adapting between model types based on different numerical algorithms, i.e. from a node-centred finite difference scheme to a cell-centred finite volume scheme. This will require careful consideration. Moreover, care must be taken when using adapters as, if many adaptations are involved, simple linear interpolation could lead to a smoothing of artefacts to such an extent that artefacts that were initially present are lost as the simulation evolves in time. An example of this problem could be specific topographic features in a two-way coupling between OTT2D and Exner.

FluidEarth 2 is targeted at Windows and Linux using .net 4.0 and Mono. Testing has been most extensive on Windows 7 but a medium level of testing has been undertaken to run Pipistrelle on Mono with a composition that has been built on Windows 7. Some cosmetic issues were found within the user interface and, at the time of writing, remain to prevent trouble-free usage on Mono. However, the main elements held up well for the compositions tried. The Mono testing environment was a virtual machine comprising: 1 CPU, 1GB RAM, 40GB HDD. The operating system installed was Linux Ubuntu 12.04.2 LTS Desktop x64. Mono was installed from the default package repository mono-complete version 2.10.8.1-1ubuntu2 monodevelop version 2.8.6.3 + dfsg-2.

The FluidEarth 2 toolkit (Pipistrelle and the FluidEarth SDK) are open source developments available on SourceForge (FluidEarth SourceForge Project 2012). The initial, 2012, versions of the code and the 2013 updates described here were developed for HR Wallingford by Adrian Harper of Innovyze. FluidEarth 2 was co-funded by the European Commission (EC) 7th Framework Programme DRIHM Project, Grant Number 283568.

REFERENCES

Anastas, P. 2010 *Agency Priority*. EPA Office of Research and Development, Washington, DC.

Becker, B. P. & Schüttrumpf, H. 2011 An OpenMI module for the groundwater flow simulation programme Feflow. *Journal of Hydroinformatics* **13** (1), 1–12.

Becker, B. P. J., Schwanenberg, D., Schruff, T. & Hatz, M. 2012 Conjunctive real time control and hydrodynamic modelling in application to rhine river. In: *10th International Conference on Hydroinformatics, HIC 2012*, Hamburg, Germany.

Braden, B. 1986 The surveyor's area formula. *The College Mathematics Journal* **17** (4), 326–337.

Bulatewicz, T., Yang, X., Peterson, J. M., Staggenborg, S., Welch, S. M. & Steward, D. R. 2010 Accessible integration of agriculture, groundwater, and economic models using the Open Modeling Interface (OpenMI): methodology and initial results. *Hydrology and Earth System Sciences* **14**, 521–534.

Cleverley, P. 2012 FluidEarth Training Website, http://eLearning.fluidearth.net (accessed December 2012).

DotSpatial Geographic Information System Library for .net4 website, http://dotspatial.codeplex.com/ (accessed October 2013).

Elag, M. & Goodall, J. L. 2011 Feedback loops and temporal misalignment in component-based hydrologic modeling. *Water Resources Research* **47** (12), W12520.

FluidEarth SourceForge Project 2012 Open Source Development for FluidEarth, http://sourceforge.net/projects/fluidearth/ (accessed 18th October 2013).

Grass, A. J. 1981 Sediment Transport by Waves and Currents. SERC London Centre of Marine Technology, Report FL29.

Gregersen, J. B., Gijsbers, P. J. A. & Westen, S. J. P. 2007 OpenMI: open modelling interface. *Journal of Hydroinformatics* **9** (3), 175–191.

Harper, A., Cleverley, P. & Kelly, D. 2012 Source Forge FluidEarth 2 Download, http://sourceforge.net/projects/fluidearth/files/latest/download (accessed December 2012).

Hubbard, M. E. & Dodd, N. 2002 A 2D numerical model of wave run-up and overtopping. *Coastal Engineering* **47**, 1–26.

Hudson, J. 2001 Numerical Techniques for Morphodynamical Modelling. PhD Thesis, Department of Mathematics, University of Reading.

Lu, B. & Piasecki, M. 2012 Community modeling systems: classification and relevance to hydrologic modelling. *Journal of Hydroinformatics* **14** (4), 840–856.

Makropoulos, C., Safiolea, E., Baki, S., Douka, E., Stamou, A. & Mimikou, M. 2010 An integrated, multi-modelling approach for the assessment of water quality: lessons from the Pinios River case in Greece. In: *Proceedings of International Environmental Modelling and Software Society (iEMSs) 2010 International Congress*, Fifth Biennial Meeting, Ottawa, Canada.

Meiburg, S. in EPA 2008 *Integrated Modeling for Integrated Environmental Decision Making*. EPA-100-R-08-010. US Environmental Protection Agency, Office of the Science Advisor, Washington, DC.

Moore, R. V. 2010 *From Google Maps to Google Models*. AGU Fall Meeting, 13–17 December, San Francisco, CA.

Moore, R. V., Gijsbers, P., Fortune, D., Gregersen, J., Blind, M., Grooss, J. & Vanecek, S. 2010 *OpenMI Document Series: Scope for the OpenMI (Version 2.0)*. Butford Technical Publishing Ltd, Pershore, UK.

Morita, M. & Yen, B. 2002 Modeling of conjunctive two-dimensional surface-three-dimensional subsurface flows. *Journal of Hydraulic Engineering* **128** (2), 184–200.

OpenMI Association Website 2012a What is OpenMI? The OpenMI Association, http://www.openmi.org/new-to-openmi#TOC-What-is-OpenMI- (accessed 15th August 2012).

OpenMI Association Website 2012b Short History? The OpenMI Association. http://www.openmi.org/new-to-openmi#TOC-Short-history (accessed 21st August 2012).

OpenMI Association Website 2010c What's New in OpenMI 2.0?, The OpenMI Association, http://www.openmi.org/learning-more#TOC-OpenMI-manuals-and-guidelines (accessed 22nd August 2012).

OpenMI Association Website 2010d OpenMI Standard 2 Specification. The OpenMI Association, https://sites.google.com/a/openmi.org/home/learning-more/OpenMIStandard2InterfaceSpecification.pdf?attredirects=0 (accessed 24th October 2013).

OpenMI SourceForge Project 2010 OpenMI, The OpenMI Association, http://sourceforge.net/projects/openmi/ (accessed 18th October 2013).

Perdreau, N. & Cunge, J. A. 1971 Sedimentation dans les estuaries et les embouchures bouchon marin et bouchon fluvial. In: *14th Congress of the IAHR*, Paris.

Safiolea, E., Baki, S., Makropoulos, C., Deliege, J. F., Magermans, P., Everbecq, E., Gkesouli, A., Stamou, A. & Mimikou, M. 2011 Integrated modelling for river basin management planning. *Proceedings of the ICE, Water Management* **164** (8), 405–419.

Shrestha, N. K., Leta, O. T., de Fraine, B., Van Griensven, A., Garcia-Armisen, T., Ouattara, N. K., Servais, P. & Bauwens, W. 2012 Integrated modelling of river Zenne using OpenMI. In: *Proceedings of 10th International Conference on Hydroinformatics, HIC 2012*, Hamburg, Germany.

Sutherland, J., Bolster, M. & Harper, A. 2013 Beachplan as an Open-MI composition. In: *Proceedings of the 35th IAHR World Congress*, Chengdu, China. In press.

Voinov, A. 2010 Model integration and the role of data. *Environmental Modelling & Software* **25**, 965–969.

First received 30 July 2013; accepted in revised form 19 November 2013. Available online 16 December 2013