OpenMI: Open modelling interface

J. B. Gregersen, P. J. A. Gijsbers and S. J. P. Westen

ABSTRACT

Management issues in many sectors of society demand integrated analysis that can be supported by integrated modelling. Since all-inclusive modelling software is difficult to achieve, and possibly even undesirable, integrated modelling requires the linkage of individual models or model components that address specific domains. Emerging from the water sector, the OpenMI has been developed with the purpose of being the glue that can link together model components from various origins. The OpenMI provides a standardized *interface* to define, describe and transfer data on a time basis between software components that run simultaneously, thus supporting systems where feedback between the modelled processes is necessary in order to achieve physically sound results. The 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. The 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.

Key words | decision support systems, integrated catchment modelling, interface standard, model linking, open source

J. B. Gregersen (corresponding author)
DHI – Water and Environment,
Agern Alle 5, DK-2970, Hørsholm,
Denmark
Tel.: +45 4516 9200
Fax: +45 4516 9292
E-mail: *gregersen@lictek.dk*

P. J. A. Gijsbers
WL – Delft Hydraulics,
PO Box 177, 2600 MH, Delft,
The Netherlands

S. J. P. Westen
WSL – Wallingford Software Ltd,
Howbery Park, WallingfordOX10 8B,
UK

INTRODUCTION

Managing environmental processes independently does not always produce sensible decisions when the wider view is taken. Therefore, it becomes important to be able to model not only the individual catchment processes – such as groundwater, river flow and irrigation – but also their interactions.

Consequently, many existing hydrological decision support systems use combined hydrological models as the main building blocks. One of the earliest of these systems was SHE, the European Hydrologic Model System (Abbott *et al.* 1986), which supports integrated modelling of surface water, unsaturated flow and groundwater flow. Since then, numerous other similar systems have been developed, where each system supports a fixed combination of specific hydrological and hydraulic models. In many cases, these systems are fulfilling the needs for integrated modelling. However, in some cases, the limited number of available combinations supported by the individual systems forces the modellers to make undesirable compromises with respect to creating an accurate representation of the physical phenomenon that is being modelled. Naturally, any system can be adapted to specific needs. Depending on the underlying software architecture this may be either difficult or fairly easy, but for most systems such tailoring requires access to the source code of the hydrological models involved. In practice, this means that such systems are typically built by model providers using only a limited suite of models for which source code is available. Even

doi: 10.2166/hydro.2007.023

when the model provider has access to a large suite of models, the number of possible useful combinations between these models means that in many cases a combination requested for a particular project is not available as an off-the-shelf product and it may not be economically feasible to create such a system for a single user or project.

The objective of the EU co-financed HarmonIT project was to address these problems through the development of an open modelling interface (the OpenMI) that will allow the easy linking of existing and new models. The HarmonIT project, led by CEH in Wallingford (UK), had 14 European partners representing end-users, research institutes and commercial model providers. Perhaps the most notable fact is that the three commercial partners (DHI – Water and Environment, WL – Delft Hydraulics and Wallingford Software), who are all providers of some of the world's most widely used modelling systems and in their normal business considered as competitors, were dedicated in sharing their knowledge and contributed in close co-operation to the development and promotion of the OpenMI standard.

Essentially, the OpenMI standard is a software component interface definition for the computational core (the engine) of the hydrological and hydraulic models. Model components that comply with this standard can, without any programming, be configured to exchange data during computation (at run-time). This means that combined systems can be created and can be based on OpenMI-compliant models from different providers, thus enabling the modeller to use those models that are best suited to a particular project. The standard supports two-way links where the involved models mutually depend on calculational results from each other. Linked models may run asynchronously with respect to timesteps and data represented on different geometries (grids) can be exchanged seamlessly.

The usefulness of the OpenMI standard relies on the availability of compliant models. In other words, when the number of relevant compliant models has reached a critical level, it becomes attractive both to deliver new compliant models and to create linked systems based on OpenMI-compliant models. Consequently, one of the main requirements for the OpenMI architecture was that it should be cost-effective to migrate models and that the architecture should, at the same time, give freedom to model providers to make their own optimal software designs. Most OpenMI-compliant models will, for many years, be based on existing models that are being migrated. Such models typically consist of thousands of lines of Fortran, C or Pascal code and re-programming is too expensive. Those models should be able to run both in their normal environment and under the OpenMI environment, without having to maintain two different versions and without having to complicate the calculation core with a great deal of OpenMI-specific code. The selected approach to fulfil all these requirements was to make a lean standard that is essentially an interface definition, allowing developers to make their own design choices. In order to separate the OpenMI-specific code from the calculation code, a wrapper design pattern was developed and a number of generic support libraries that can speed up the migration process were developed.

The ambition for OpenMI is that the standard should be accepted and used by a wide range of model developers and model users and possibly become a new world standard for model linking. Much effort was put into making comprehensive documentation and guidelines (Gijsbers *et al.* 2005; Moore *et al.* 2005; Tindall *et al.* 2005) and the standard and all tools and libraries were made available as open source (SourceFORGE 2005). Undoubtedly, there will be requests for improvement of the OpenMI standard when the larger community starts using it. In order to meet such requirements an OpenMI association is currently being established. This association will be open for everyone to join and will be responsible for the maintenance and further development of the standard.

In order to demonstrate the capabilities of the OpenMI a complex system of integrated catchment modelling is shown in Figure 1. Meteorological data from a number of measurement stations are handled by a database system. This system will provide precipitation and evaporation data to rainfall–runoff models. The rivers are modelled by a simple conceptual river model that will obtain inflow data from the rainfall–runoff models. For a particular river reach, a more detailed representation of the river flow is required. This river reach is modelled by a physically based hydrodynamic river model. The river model will obtain inflow data for its upper boundary from the conceptual river model and will provide inflow data for the conceptual

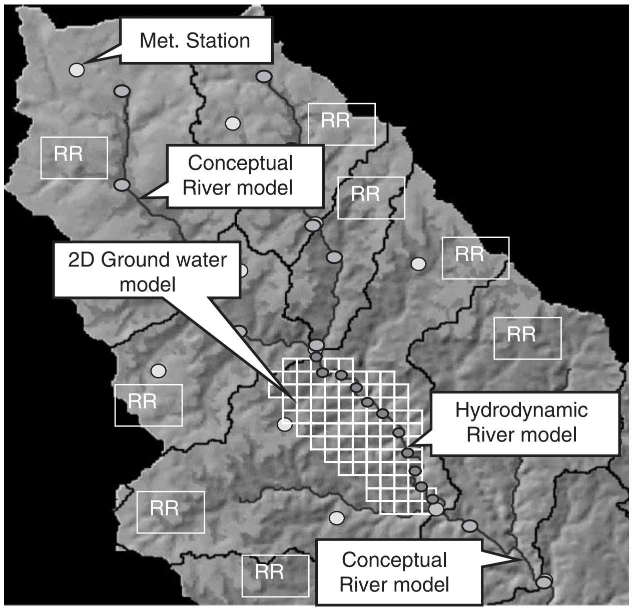


Figure 1 | Example of integrated catchment modelling that can be realized by combining OpenMI-compliant models.

river models that connect to the downstream boundary of the hydrodynamic river model. Interaction between groundwater and surface water is considered important at the location of the hydrodynamic model. The underlying aquifer is modelled by a 2D distributed groundwater model. This groundwater model will receive leakage from the river model. The river model will calculate this leakage based on information about the groundwater level, which is obtained from the groundwater model.

A model user who has access to appropriate OpenMI-compliant models can establish such a system. The procedure the user needs to follow in order to establish the system is as follows:

1. The user populates each of the models with the required data through the preparatory user interfaces of each individual model. The prepared input data for these models is saved to disk. Most OpenMI models will, when input files are saved, also create a small OpenMI standardized XML file (the OMI file), which contains information about the filename for the OpenMI-compliant model component (the LinkableComponent) and the filenames for input files.
2. The user uses an OpenMI editor to add the models to a configuration. A screen dump for the open source OpenMI configuration editor is shown in Figure 2. In practice, the user selects [Add model] in the configuration editor and browses the file system to find the OMI files. Each time an OMI file has been selected, the OpenMI-compliant model component is loaded and the component reads its input files.
3. Connections between the models are created simply by dragging arrows from one model to another. Clicking a connection brings up a link editor dialog (see Figure 3). This dialog has information, obtained from the model components, about which quantities can be provided and which quantities can be accepted. The user can then select the required combination.
4. When all the links have been established the user can run the linked system.
5. When the model run has completed, the user can investigate the results from the calculations through the proprietory user interfaces of each individual model.

The steps described above are valid for the open source user interface. The OpenMI standard does not prescribe that this particular user interface should be used, and we foresee that in the future there will be many different user interfaces available.

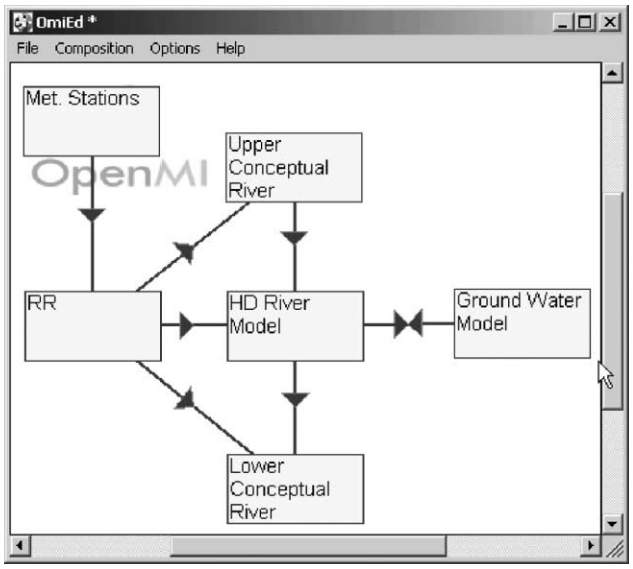
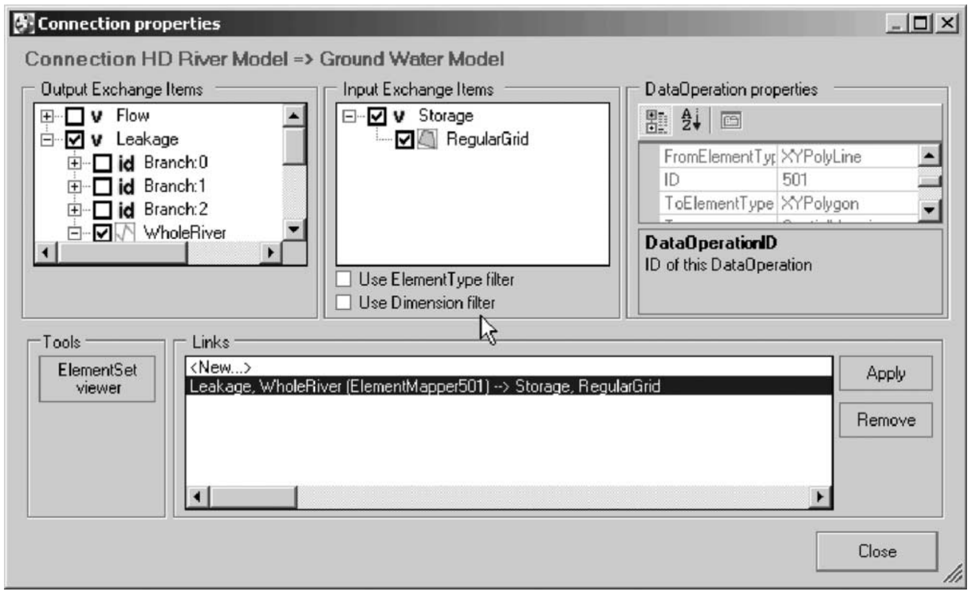


Figure 2 | Open source OpenMI configuration editor used to configure links between the models shown in Figure 1.



Connection HD River Model => Ground Water Model

Output Exchange Items: Flow; [✓ Leakage]; Branch:0; Branch:1; Branch:2; [✓ WholeRiver]

Input Exchange Items: [✓ Storage]; [✓ RegularGrid]

Use ElementType filter

Use Dimension filter

FromElementTyp	XYPolyLine
ID	501
ToElementType	XYPolygon

DataOperationID
ID of this DataOperation

<New...>
Leakage, WholeRiver (ElementMapper501) --> Storage, RegularGrid

Figure 3 | Open source OpenMI configuration editor's link property dialog for the link from the hydrodynamic river model to the groundwater model.

EXISTING MODEL SYSTEMS

Before going into detail about the OpenMI, some definitions of the existing model systems are given.

A *model application* is the entire model software system that is installed on a computer. Normally a model application consists of a user interface and an engine. The *engine* is where the calculations take place. The user supplies information through the user interface, which generates input files for the engine. The user can run the model simulation, for example by pressing a button in the user interface that deploys the engine (see Figure 4). The engine reads the input files, perform calculations and finally writes the results to output files.

When an engine has read its input files it becomes a *model*. In other words, a model is an engine populated with data. A model can simulate the behaviour of a specific physical entity (e.g. the River Rhine). If an engine can be instantiated separately and has a well-defined interface it becomes an *engine component*. An engine component populated with data is a *model component*.

There are many variations of the model application pattern described above but most important from the OpenMI perspective are the distinctions between model application, engine, model, engine component and model component.

THE OPENMI

The OpenMI is based on the 'request & reply' mechanism. The OpenMI is a pull-based pipe-and-filter architecture, which consists of communicating components (source components and target components) that exchange memory-based data in a predefined way and in a predefined format. The OpenMI defines the component interfaces, as

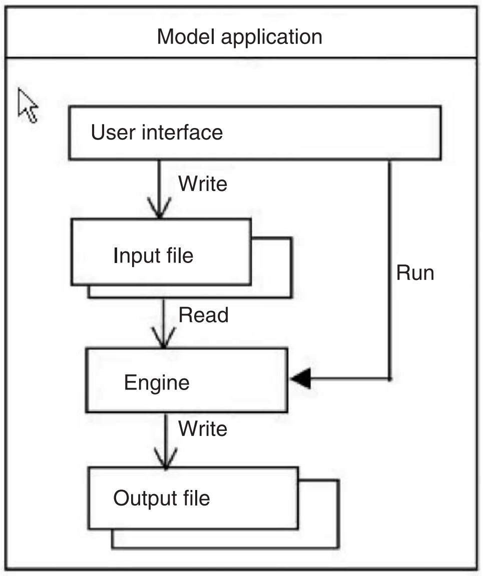


Figure 4 | Existing model application pattern.

well as how the data is being exchanged. The components in the OpenMI are called *linkable components* to indicate that it involves components that can be linked together. From the data exchange perspective, the OpenMI is a purely single-threaded architecture, where an instance of a linkable component handles only one data request at a time before acting upon another request. Data exchange in the OpenMI architecture is triggered by a component at the end of the component chain. Once triggered, components exchange data autonomously without any supervising authority. If necessary, components start their own computing process to produce the requested data. Most important, however, is the fact that the OpenMI is not based on a framework; it only has linkable components.

LinkableComponent

Essentially, a model can be regarded as an entity that can provide data and/or accept data. Most models receive data by reading input files and provide data by writing output files. However, the approach for the OpenMI is to access the model directly at run-time and not to use files for data exchange. In order to make this possible, the engine needs to be turned into an engine component and the engine component needs to implement an interface through which the data inside the component is accessible. The OpenMI defines a standard interface for engine components (ILinkableComponent – see Figure 5) that OpenMI-compliant engine components must implement. When an engine component implements the ILinkableComponent interface it becomes an OpenMI LinkableComponent.

Link (what is exchanged)

One LinkableComponent can retrieve data from another LinkableComponent by invocation of the GetValues method. However, this is only possible if the two components have information about each other's existence and have a clear definition of the kind of data that has been requested. This information is contained in a class that implements the OpenMI ILink interface. Before invocation of the GetValues method a Link object must be created, populated and added to the two components by use of the AddLink method. The Link object holds a reference (handle) to the two linked components. The Link object also contains information about *what* is requested, *where* the requested values apply and *how* the requested data should be calculated. This information is included in the OpenMI Quantity class, the OpenMI ElementSet class and the OpenMI DataOperation class, respectively (see the ILink interface in Figure 5). The Link class defines a specific connection between two LinkableComponents. For two specific LinkableComponents many possible links may exist.

Quantity (what)

The Quantity object defines what should be retrieved. This could be water level or flow, for example. The Quantity class represents this information simply as a text string (the Description property). OpenMI does not provide any naming convention for quantities. The Link class has a target Quantity object and a source Quantity object. The quantity description in the source Quantity object must be recognizable by the source LinkableComponent and the quantity description in the target Quantity object must be recognizable by the target LinkableComponent. It is the responsibility of the person who configures the linked system to ensure that the combination of the two particular quantities makes sense physically.

ElementSet (where)

The ElementSet object defines where the retrieved values must apply. For example, a groundwater model may be asked for either the groundwater level at a particular point or the groundwater level as an average value over a polygon; a river model may be asked for the flow at a particular calculation node. These locations are defined in the ElementSet. The ElementSet is a collection of Elements, where each element can be an ID-based entity, like a particular node, or a geometrical entity. A geometrical entity is a point, a polyline, a polygon or a polyhedron. The GetValues method returns a ValueSet, which is an array of values or an array of vectors. Each value or vector in the returned ValueSet applies to one Element in the target ElementSet.

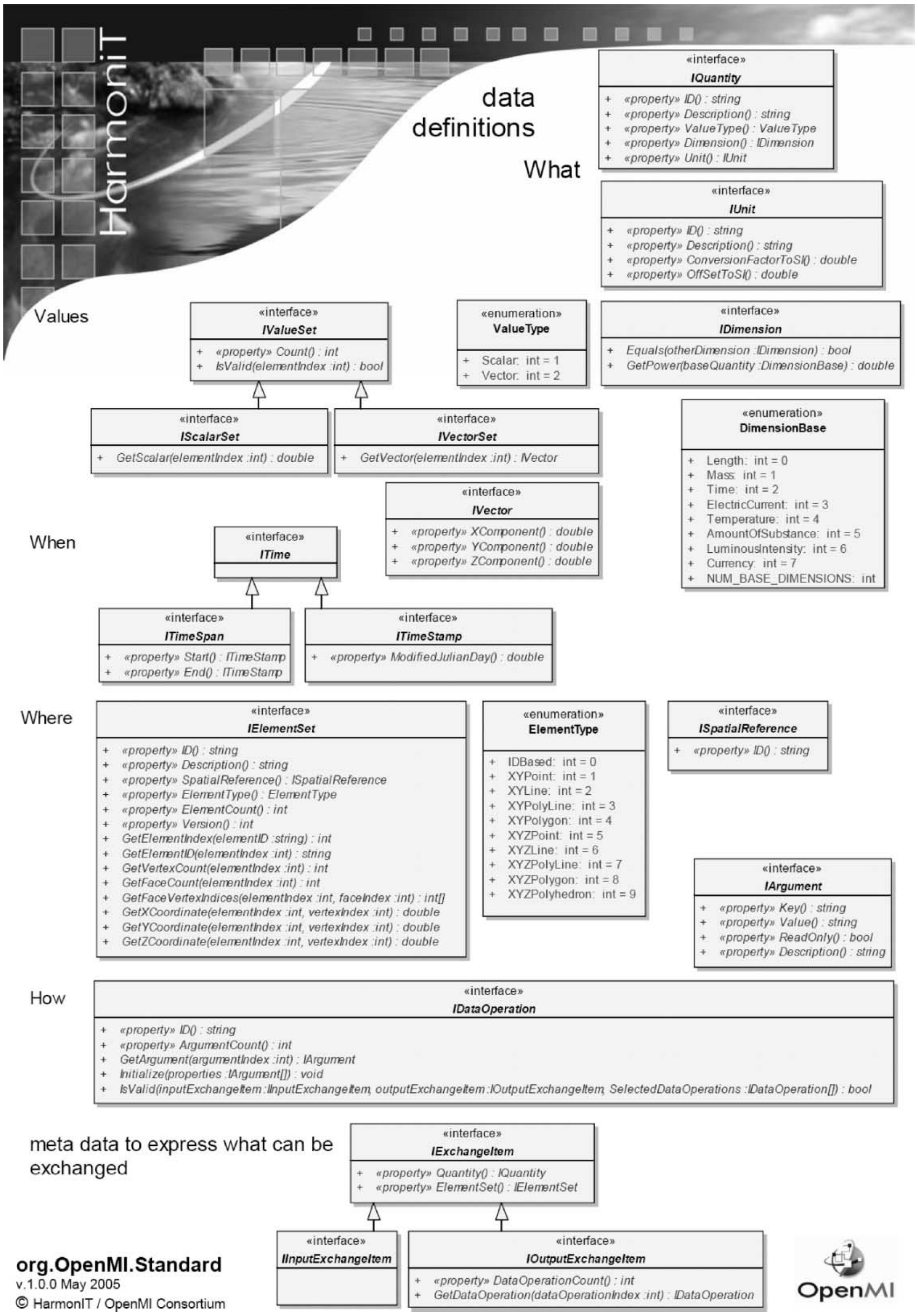
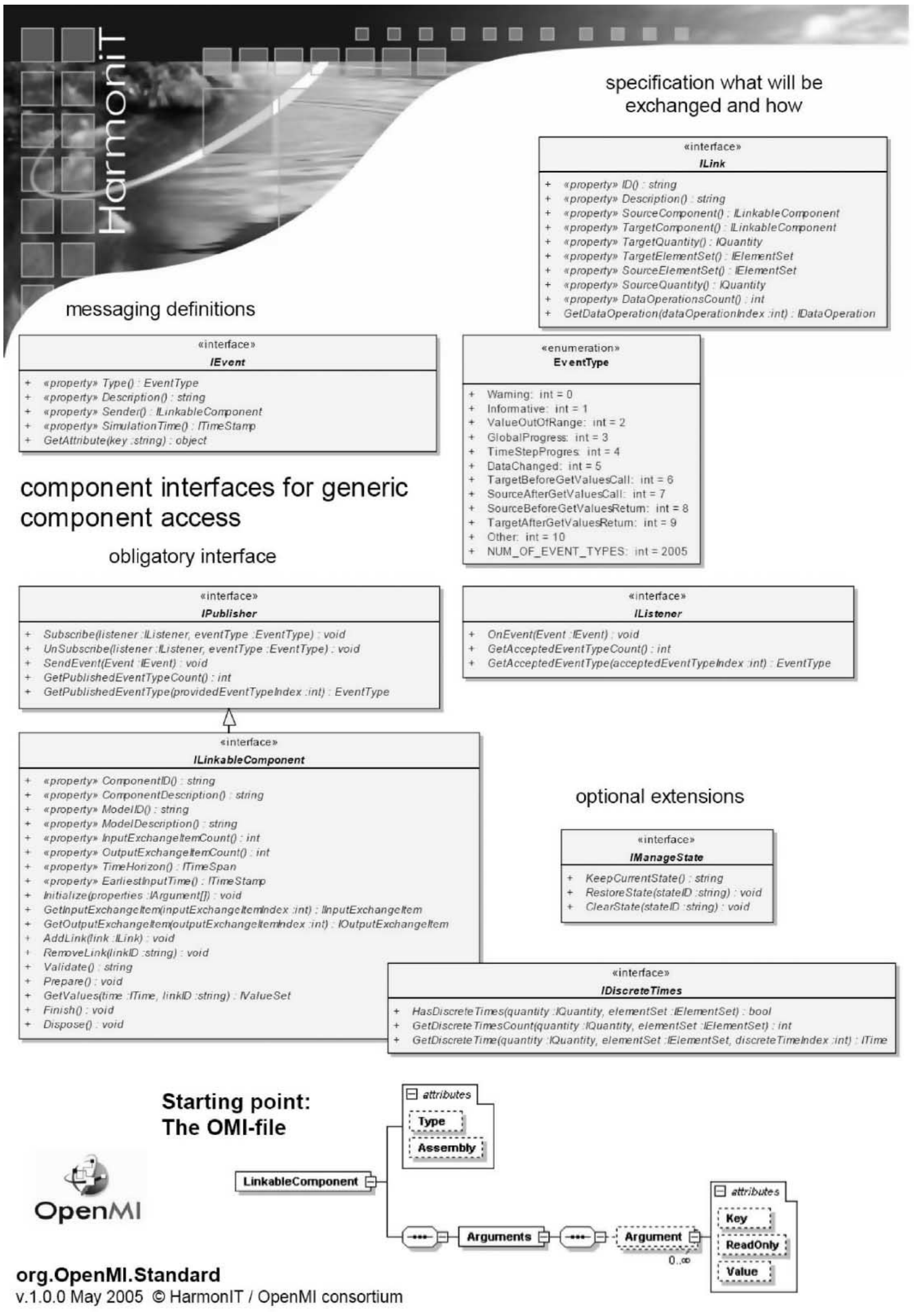

HarmoniT
data definitions
What
«interface» IQuantity
+ «property» ID() : string
+ «property» Description() : string
+ «property» ValueType() : ValueType
+ «property» Dimension() : IDimension
+ «property» Unit() : IUnit
«interface» IUnit
+ «property» ID() : string
+ «property» Description() : string
+ «property» ConversionFactorToSI() : double
+ «property» OffSetToSI() : double
Values
«interface» IValueSet
+ «property» Count() : int
+ IsValid(elementIndex :int) : bool
«enumeration» ValueType
+ Scalar: int = 1
+ Vector: int = 2
«interface» IDimension
+ Equals(otherDimension :IDimension) : bool
+ GetPower(baseQuantity :DimensionBase) : double
«interface» IScalarSet
+ GetScalar(elementIndex :int) : double
«interface» IVectorSet
+ GetVector(elementIndex :int) : IVector
«enumeration» DimensionBase
+ Length: int = 0
+ Mass: int = 1
+ Time: int = 2
+ ElectricCurrent: int = 3
+ Temperature: int = 4
+ AmountOfSubstance: int = 5
+ LuminousIntensity: int = 6
+ Currency: int = 7
+ NUM_BASE_DIMENSIONS: int
«interface» IVector
+ «property» XComponent() : double
+ «property» YComponent() : double
+ «property» ZComponent() : double
When
«interface» ITime
«interface» ITimeSpan
+ «property» Start() : ITimeStamp
+ «property» End() : ITimeStamp
«interface» ITimeStamp
+ «property» ModifiedJulianDay() : double
Where
«interface» IElementSet
+ «property» ID() : string
+ «property» Description() : string
+ «property» SpatialReference() : ISpatialReference
+ «property» ElementType() : ElementType
+ «property» ElementCount() : int
+ «property» Version() : int
+ GetElementIndex(elementID :string) : int
+ GetElementID(elementIndex :int) : string
+ GetVertexCount(elementIndex :int) : int
+ GetFaceCount(elementIndex :int) : int
+ GetFaceVertexIndices(elementIndex :int, faceIndex :int) : int[]
+ GetXCoordinate(elementIndex :int, vertexIndex :int) : double
+ GetYCoordinate(elementIndex :int, vertexIndex :int) : double
+ GetZCoordinate(elementIndex :int, vertexIndex :int) : double
«enumeration» ElementType
+ IDBased: int = 0
+ XYPoint: int = 1
+ XYLine: int = 2
+ XYPolyLine: int = 3
+ XYPolygon: int = 4
+ XYZPoint: int = 5
+ XYZLine: int = 6
+ XYZPolyLine: int = 7
+ XYZPolygon: int = 8
+ XYZPolyhedron: int = 9
«interface» ISpatialReference
+ «property» ID() : string
«interface» IArgument
+ «property» Key() : string
+ «property» Value() : string
+ «property» ReadOnly() : bool
+ «property» Description() : string
How
«interface» IDataOperation
+ «property» ID() : string
+ «property» ArgumentCount() : int
+ GetArgument(argumentIndex :int) : IArgument
+ Initialize(properties :IArgument[]) : void
+ IsValid(inputExchangeItem :IInputExchangeItem, outputExchangeItem :IOutputExchangeItem, SelectedDataOperations :IDataOperation[]) : bool
meta data to express what can be exchanged
«interface» IExchangeItem
+ «property» Quantity() : IQuantity
+ «property» ElementSet() : IElementSet
«interface» IInputExchangeItem
«interface» IOutputExchangeItem
+ «property» DataOperationCount() : int
+ GetDataOperation(dataOperationIndex :int) : IDataOperation
org.OpenMI.Standard
v.1.0.0 May 2005
© HarmonIT / OpenMI Consortium
OpenMI

Figure 5 | *org.OpenMI.Standard interfaces.*



```
«interface»
ILink
+ «property» ID() : string
+ «property» Description() : string
+ «property» SourceComponent() : ILinkableComponent
+ «property» TargetComponent() : ILinkableComponent
+ «property» TargetQuantity() : IQuantity
+ «property» TargetElementSet() : IElementSet
+ «property» SourceElementSet() : IElementSet
+ «property» SourceQuantity() : IQuantity
+ «property» DataOperationsCount() : int
+ GetDataOperation(dataOperationIndex :int) : IDataOperation
```

```
«interface»
IEvent
+ «property» Type() : EventType
+ «property» Description() : string
+ «property» Sender() : ILinkableComponent
+ «property» SimulationTime() : ITimeStamp
+ GetAttribute(key :string) : object
```

```
«enumeration»
EventType
+ Warning: int = 0
+ Informative: int = 1
+ ValueOutOfRange: int = 2
+ GlobalProgress: int = 3
+ TimeStepProgres: int = 4
+ DataChanged: int = 5
+ TargetBeforeGetValuesCall: int = 6
+ SourceAfterGetValuesCall: int = 7
+ SourceBeforeGetValuesReturn: int = 8
+ TargetAfterGetValuesReturn: int = 9
+ Other: int = 10
+ NUM_OF_EVENT_TYPES: int = 2005
```

```
«interface»
IPublisher
+ Subscribe(listener :IListener, eventType :EventType) : void
+ UnSubscribe(listener :IListener, eventType :EventType) : void
+ SendEvent(Event :IEvent) : void
+ GetPublishedEventTypeCount() : int
+ GetPublishedEventType(providedEventTypeIndex :int) : EventType
```

```
«interface»
IListener
+ OnEvent(Event :IEvent) : void
+ GetAcceptedEventTypeCount() : int
+ GetAcceptedEventType(acceptedEventTypeIndex :int) : EventType
```

```
«interface»
ILinkableComponent
+ «property» ComponentID() : string
+ «property» ComponentDescription() : string
+ «property» ModelID() : string
+ «property» ModelDescription() : string
+ «property» InputExchangeItemCount() : int
+ «property» OutputExchangeItemCount() : int
+ «property» TimeHorizon() : ITimeSpan
+ «property» EarliestInputTime() : ITimeStamp
+ Initialize(properties :IArgument[]) : void
+ GetInputExchangeItem(inputExchangeItemIndex :int) : IInputExchangeItem
+ GetOutputExchangeItem(outputExchangeItemIndex :int) : IOutputExchangeItem
+ AddLink(link :ILink) : void
+ RemoveLink(linkID :string) : void
+ Validate() : string
+ Prepare() : void
+ GetValues(time :ITime, linkID :string) : IValueSet
+ Finish() : void
+ Dispose() : void
```

```
«interface»
IManageState
+ KeepCurrentState() : string
+ RestoreState(stateID :string) : void
+ ClearState(stateID :string) : void
```

```
«interface»
IDiscreteTimes
+ HasDiscreteTimes(quantity :IQuantity, elementSet :IElementSet) : bool
+ GetDiscreteTimesCount(quantity :IQuantity, elementSet :IElementSet) : int
+ GetDiscreteTime(quantity :IQuantity, elementSet :IElementSet, discreteTimeIndex :int) : ITime
```

Figure 5 | *continued.*

DataOperation (how)

The DataOperation object defines how the requested values should be calculated. Examples of data operations are time-accumulated, spatially averaged and maximum values. There are no OpenMI conventions for data operations. As for quantities, data operations are simply defined by a text string, which is recognizable by the source LinkableComponent.

ExchangeItem (what can be exchanged)

When model links are created and populated, information about which quantities, locations and data operations each LinkableComponent supports is needed. This information can be obtained by querying the LinkableComponents for InputExchangeItems and OutputExchangeItems. Each InputExchangeItem contains a Quantity and an ElementSet, describing what can be accepted at which location. Each OutputExchangeItem contains a Quantity and an ElementSet, describing what can be provided at which location. OutputExchangeItems also contain information about available DataOperations. OpenMI configuration editors (see Figure 3) typically query the ILinkableComponent interfaces in order to display potential input and output exchange items to the user for each model in a configuration. This enables the user to configure and establish the required connections (Links).

Time

Time in OpenMI is defined by either a TimeStamp or a TimeSpan. A time stamp is a single point in time, whereas a time span is a period from a begin time to an end time. Each of these times is represented by the Modified Julian Date. A modified Julian date is the Julian date minus 2400000.5 and represents the number of days since midnight November 17, 1858 Universal Time in the Julian calendar.

GetValues

Now let us move back to the essence of the OpenMI, the GetValues method. When one LinkableComponent invokes the GetValues method of another LinkableComponent, the source LinkableComponent must return the values for the specified quantity, at the specified time stamp or time span and at the specified location. If the LinkableComponent is of the time-stepping kind of numerical model it does no calculation until it receives a GetValues call. When the GetValues method is invoked, the LinkableComponent calculates as long as it is necessary to obtain the needed data. Usually it is necessary for the source component to interpolate or extrapolate its internal data in time and space before these can be returned.

The OpenMI architecture puts a lot of responsibilities on LinkableComponents. One of the reasons for this is that we feel that any data conversion, like interpolations, can be done in the most optimal way by the source component. If the source component is a groundwater model, for example, any interpolations of the groundwater levels are most safely done by the groundwater model itself rather than some external tool.

The OpenMI framework is very simple – or you may say that there is no framework. All there is are LinkableComponents. Once the system of linked model components is created, the invocation of GetValues methods from one model component to another is driving the calculations. Since the ILinkableComponent interface (Figure 5) does not have any methods that can be used to start the chain of calculations, a trigger component is needed. The trigger component is a LinkableComponent that has an additional method for starting calculations (see the example below).

EXAMPLE

Let us look at a very simple example. A conceptual lumped rainfall–runoff (RR) model provides inflow to a river model. The populated link class is shown in Figure 6. Note that this link is 'ID-based' because the ElementSets are not populated with any information about the spatial representation of the models. Consequently, no spatial operations will be performed when data is exchanged.

The sequence diagram in Figure 7 shows the calling sequence for a configuration with a river model linked to a rainfall–runoff model. The functionality shown in Figure 7 will typically be implemented in an OpenMI user interface like the one shown in Figures 2 and 3.

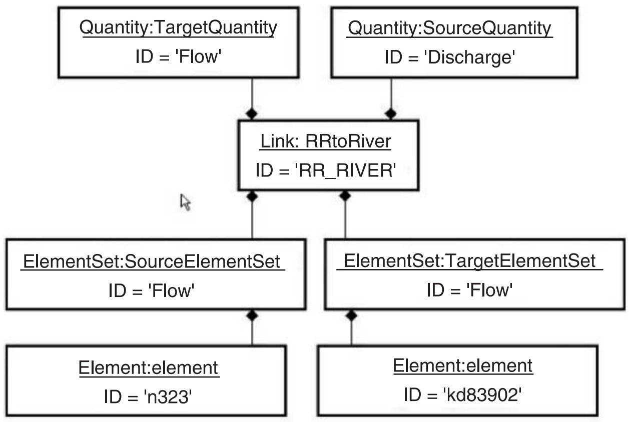


Figure 6 | The populated Link class.

The sequence diagram has the following steps:

1. The River Model object and the RR Model object are instantiated. The Initialize method is then invoked for both objects. Models will typically read their private input files when the Initialize methods is invoked. Information about name and location of the input files can be passed as arguments in the Initialize method.
2. The River Model is queried for InputExchangeItems and the RR Model is queried for OutputExchangeItems. The InputExchangeItems and OutputExchangeItems objects contain information about which combinations of Quantities and ElementSets (locations) can be accepted by the components as input or output, respectively.
3. A Link object is created and populated based on the obtained lists of InputExchangeItems and OutputExchangeItems. This example uses a hard-coded configuration. However, if a configuration editor were used the OutputExchangeItems and the InputExchangeItems would be selected by the user from a selection box, for example (see Figure 3).
4. The trigger component is created. This component is a very simple LinkableComponent whose only purpose is to trigger the calculation chain.
5. Link objects are added to the LinkableComponents. This will enable the LinkableComponents to invoke the GetValues method in the LinkableComponent to which they are linked.
6. The Prepare method is invoked in all LinkableComponents. This will make each LinkableComponent do whatever preparations are needed before calculations can start.
7. The RunSimulation method is invoked in the trigger object to start the calculation chain.
8. The trigger object invokes the GetValues method in the River Model and the River Model calculates until it has reached the EndTime specified in the argument list.
9. Before the River Model can complete a timestep it must update its inflow boundary condition. In order to do this, the GetValues method in the RR Model is invoked.
10. The RR Model repeatedly performs timesteps until it has reached or exceeded the time for which it was requested. If the River Model and the RR Model are not synchronous with respect to timesteps, the RR Model must interpolate the calculated runoff in time before the values can be returned.
11. The River Model has now obtained its inflow boundary value and can perform a timestep. The River Model repeatedly invokes the GetValues method in the RR Model and perform timesteps until it has reached or exceeded the EndTime, whereafter it returns control and values to the trigger object.
12. The trigger object returns control to the main program.
13. The main program invokes the Finish and Dispose methods in all LinkableComponents. LinkableComponents will typically close output files when the Finish method is invoked. The Dispose method will usually be used by the LinkableComponents to de-allocate memory.

SPATIAL MAPPING

Hydrological models usually have a schematization or grid that represents the spatial resolution of the model. For groundwater models, regular or non-regular two- or three-dimensional grids may be used, whereas river models typically use a one-dimensional grid. Conceptual catchment models may use a closed polygon to describe the catchment boundary.

When models with different spatial schematizations are linked, the values associated with one schematization in the source model must be transformed to be represented on the schematization of the target model. In order to make such transformations possible for any combination of models a

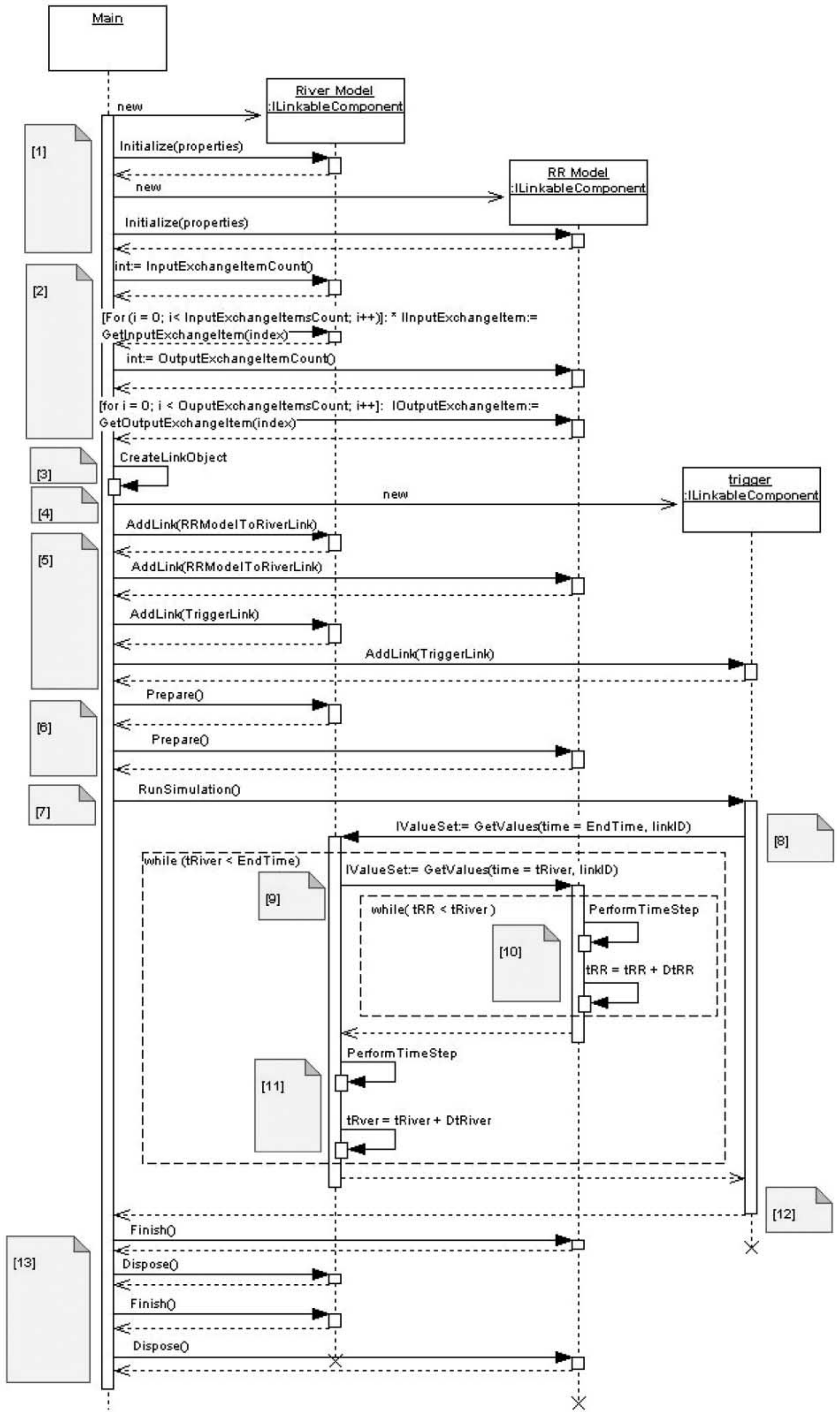

Main
River Model :ILinkableComponent
RR Model :ILinkableComponent
trigger :ILinkableComponent
new
Initialize(properties)
int:= InputExchangeItemCount()
[For (i = 0; i< InputExchangeItemsCount; i++)]: * IInputExchangeItem:= GetInputExchangeItem(index)
int:= OutputExchangeItemCount()
[for i = 0; i < OuputExchangeItemsCount; i++]: IOutputExchangeItem:= GetOutputExchangeItem(index)
CreateLinkObject
AddLink(RRModelToRiverLink)
AddLink(TriggerLink)
Prepare()
RunSimulation()
IValueSet:= GetValues(time = EndTime, linkID)
while (tRiver < EndTime)
IValueSet:= GetValues(time = tRiver, linkID)
while(tRR < tRiver)
PerformTimeStep
tRR = tRR + DtRR
tRver = tRiver + DtRiver
Finish()
Dispose()
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
[13]

Figure 7 | Configuration and deployment of linked models.

standardized spatial representation was defined. In OpenMI terms this is called the IElementSet interface (see Figure 5).

A two-dimensional grid for a groundwater model and a one-dimensional grid for a river model are shown in Figure 8.

The groundwater model grid consists of four elements GE_1, GE_2, GE_3 and GE_4. These elements have the type *Polygon*. Each corner of these polygons is a Vertex, and each Vertex has Co-ordinates. The type Polygon is defined in the standard in the enumeration ElementType, whereas there are no interface definitions for Vertex and Co-ordinate; these are only part of the OpenMI terminology and are used in the naming of some of the methods in the IElementSet interface.

The grid for the river model can be represented by an implementation of the IElementSet interface, where each branch is an element of type *Polyline*. Each Polyline element will have two vertices, one at each end of the line. The elements for the river model are shown in Figure 8 as RE_1, RE_2 and RE_3.

Assume that the river model and the groundwater model illustrated in Figure 8 are linked for the purpose of transferring information about groundwater leakage from the river model to the groundwater model. When the groundwater model invokes the GetValues method in the river model in order to obtain the leakages, the river model must return a ValueSet, where each value represents the leakage that enters each element in the groundwater model for the requested time or timespan.

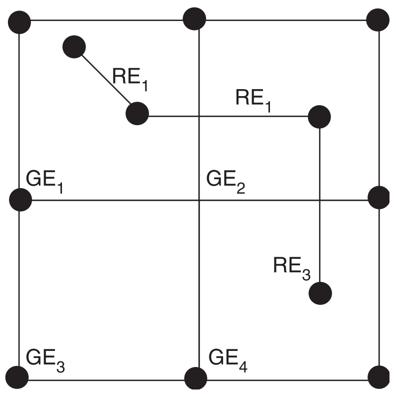


Figure 8 | Two-dimensional groundwater model grid and one-dimensional river model grid.

With this in mind it may seem like a huge task to implement the GetValues method in the linkable component. However, since all spatial representations can be accessed generically through the IElementSet interface, a generic element mapper could be developed. Such a mapper is provided in the open source org. OpenMI.Utilities.Spatial package.

The two most essential methods in the ElementMapper class are:

void Initialize(string method, IElementSet fromElements, IElementSet toElements)

and

IValueSet MapValues(IValueSet inputValues)

When Initialize is invoked, an internal mapping matrix is created. This is typically done only once before the calculations starts.

During calculations, when the GetValues method is invoked, the source component uses this mapping matrix to make the spatial conversion, simply by multiplying the vector of values associated with the grid of the source components with the mapping matrix.

For the example shown in Figure 8, the mapping matrix will look as shown below:

$$\mathbf{A} = \begin{pmatrix} 1 & 1/3 & 0 \\ 0 & 2/3 & 1/2 \\ 0 & 0 & 0 \\ 0 & 0 & 1/2 \end{pmatrix}. \tag{1}$$

When the groundwater model invokes the GetValues method in the river model, the river model can make the spatial transformation of its internal calculated leakages using the following multiplication:

$$\boldsymbol{I} = \boldsymbol{L} \times \boldsymbol{A} \tag{2}$$

where $\boldsymbol{I}$ is a vector with four components describing the leakage contribution to each grid cell in the groundwater model and $\boldsymbol{L}$ is a vector with three components, each value being the calculated leakage in a river branch.

It is important to note that element mapping as described above has nothing to do with the OpenMI standard. Anyone can implement the transformations as they please; for example, if the linkable component is an analytic model, and therefore has no grid element, mapping does not make sense.

BI-DIRECTIONAL LINKS

The simple example describes a link between a rainfall–runoff model and a river model. The calculated runoff does not depend on the conditions of the river. However, there are many examples of model linkages where information needs to go both ways. For example, if two physically based hydrodynamic river models are linked, the flow rate from the upper river to the lower river depends on the water level at the connection point in the lower river, and this water level obviously depends on the inflow from the upper river. Another example could be linkages between a groundwater model and a river model, as described in the example above, which usually also require a two-way exchange of information. The leakage rate from the river depends on the groundwater level and the groundwater level depends on the leakage rate from the river. In order to accommodate such two-way dependences two links need to be established. In the example of the groundwater model and the river model, there is one link where the river model is the source component and delivering leakage and another where the groundwater model is the source component and delivering groundwater level.

Having two such links in the configuration leads to a risk of deadlocks, where the two models keep asking each other for data without progressing in time. To avoid such deadlocks, the OpenMI standard requires that: (1) a linkable component may not invoke GetValues in other components before any previous invocations of GetValues from this component have returned and (2) a linkable component must always return values when the GetValues method is invoked. This means that bi-directional linked models may need to return extrapolated values because they cannot progress in time due to the first restriction.

The mechanisms described above can be implemented in the linkable components by setting an internal flag (e.g. isBusy) to True when the GetValues methods is invoked in another component. When the values from the other component are returned the flag will be changed back to False. If the GetValues method is invoked in a LinkableComponent which *is busy*, this component will not progress in time in order to obtain new values but will return values based on already calculated values, for example using extrapolation.

The communication between a river model and a groundwater model is illustrated in Figure 9. The steps in the sequence diagram are as follows:

1. The trigger component invokes GetValues in the groundwater model in order to make the models run until time tT.
2. The groundwater model cannot progress until it has obtained the leakages from the river model. In order to avoid deadlock situations the groundwater model sets the internal variable isBusy to True.
3. The groundwater model invokes GetValues in the river model in order to obtain the average leakages for each groundwater grid element for the period corresponding to its internal next timestep ([tG, tG–dtG]).
4. The river model sets its internal status to busy in order to avoid deadlocks. This will not have any impact in this particular configuration. However, if the river model is used in other configurations this action may be needed. Consequently, model components should always set the internal status to busy before invocation of GetValues.
5. The river model invokes GetValues in the groundwater model in order to obtain the groundwater level at the location of each river branch element for the time corresponding to the end of the next timestep (tR + dtR).
6. Since the groundwater model is busy, it cannot invoke GetValues in the river model, which means that it cannot perform timesteps. Consequently, the returned groundwater levels must be calculated by means of extrapolation, based on previous calculated values.
7. The river model changes its busy status to False, performs a timestep and increments the internal time.
8. Steps 4–7 are repeated until the internal time of the river model has exceeded the time period for which values were requested (tG + dtG).
9. Leakages are returned.

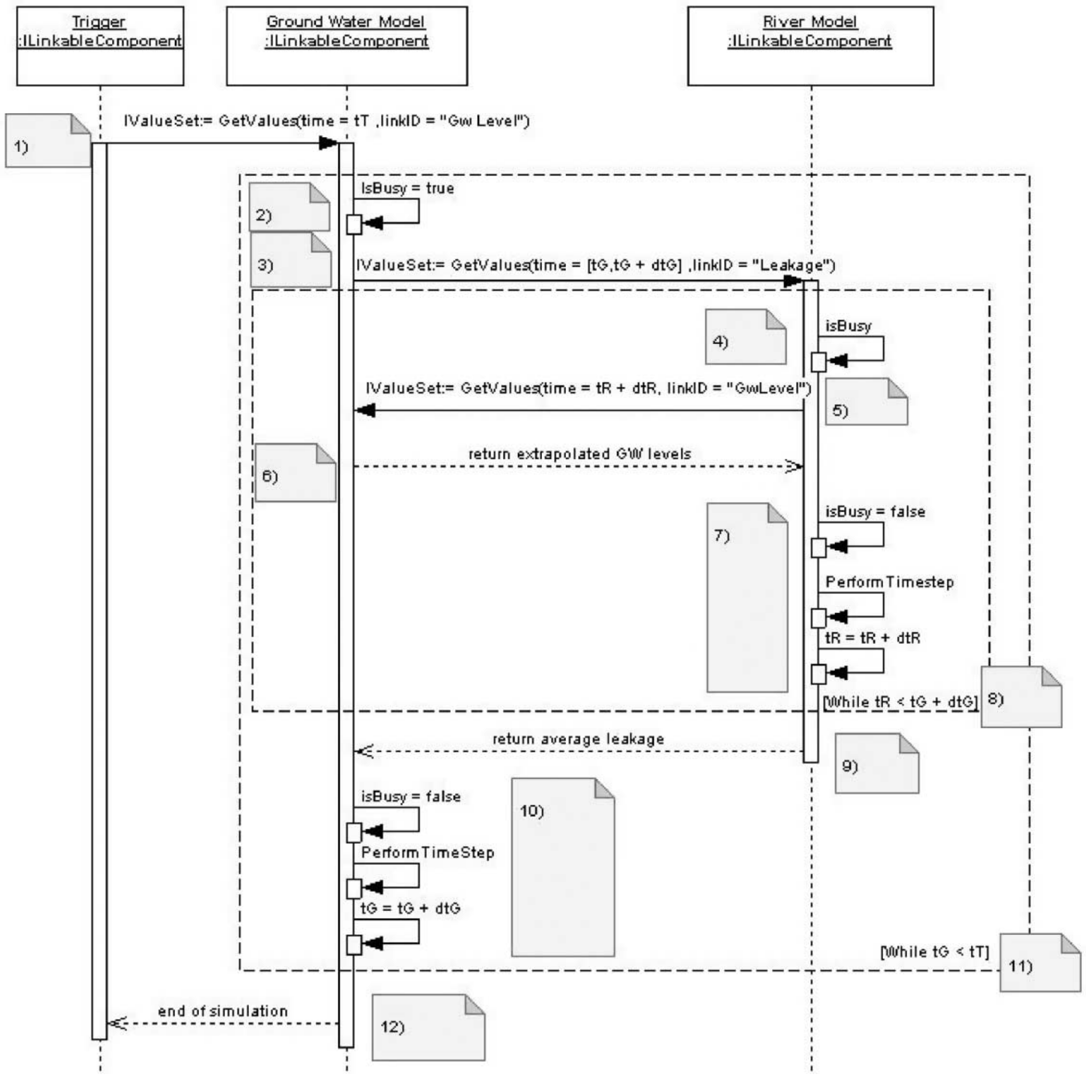


Figure 9 | Bi-directional linked groundwater model and river model.

10. The groundwater model changes its busy status to False, performs a timestep and increments its internal time.
11. Steps 2–10 are repeated until the internal time of the groundwater model has exceeded the time period for which values were requested by the trigger component.
12. Final return and end of the simulation.

Most river models and groundwater models will use an implicit numerical scheme internally. The link between the two models can be regarded as semi-explicit because the values passed from the river model to the groundwater model are based on real calculations whereas the values passed from the groundwater model to the river model are extrapolated. Extrapolations are performed by the providing component, which will enable the model component provider to use extrapolation methods optimized for the particular domain of the model (in this case, groundwater levels).

In some cases bi-directional linked systems may be subject to numerical instabilities, numerical errors or mass balance errors. The results from the simulation may also change depending on the component to which the trigger is linked. Consequently, configuring bi-directional linked systems requires in-depth knowledge of the underlying physics and numerics of the linked models. Adjusting the internal timestep length of the models, selecting the best trigger component and selecting the best extrapolation method are means of improving the results.

There will be situations where simple linking is inadequate. For such situations, an iterative configuration can be used. The OpenMI standard facilitates the creation of control components, such as iteration controllers. Control components are LinkableComponents (implementing the ILinkableComponent interface), which means that configurations including iteration controllers can be created using the OpenMI configuration editor, for example. LinkableComponents used with iteration controllers need to implement one additional interface – the IManageState interface (see Figure 5). The IManageState interface has methods for saving and restoring states, thus enabling an iteration controller to re-run a LinkableComponent from a previously saved state. Details of control components are given in the OpenMI Guidelines (Tindall *et al.* 2005).

MODEL MIGRATION

The OpenMI standard is basically a collection of interface definitions. In order to make real model engines run and exchange data these interfaces must be implemented using a programming language. Within the HarmonIT project these interfaces were defined in C# (.Net). A default implementation of each interface was also developed in a package called OpenMI backbone and a utility package was developed to assist the migration of existing model engines. All implementations are available as open source (SourceFORGE 2005).

Existing model applications typically follow the pattern shown in Figure 4. The calculation core – the engine – is typically implemented in Fortran, Pascal or C and compiled to one executable file. Such engines may consist of thousands of lines of code and many years may have been spent on development and testing. Consequently, re-implementation of such engines is not an option.

With this in mind we had, from the very beginning of the development of the OpenMI standard, three important requirements: (1) it must be easy to migrate existing models (the time for migrating a model should be less than two working months for even the most complicated models), (2) the same model engine must be able to run both in its normal environment and in the OpenMI environment and (3) the amount of OpenMI-specific implementations in the engine core should be as small as possible.

In order to meet these requirements an OpenMI wrapper package was implemented to assist in model migration. This package is implemented in C# and is available as open source (SourceFORGE 2005). The wrapper package is not part of the OpenMI standard and consequently not required in order to create OpenMI-compliant models. In this way the OpenMI standard remains simple and leaves much freedom for those migrating models to make their own design choices; at the same time, the standard makes it easy to migrate a model for those who choose to use the utilities packages.

The wrapper package provides a default implementation of the ILinkableComponent (class LinkableEngine) interface, specifically aimed at the time-stepping type of model. The wrapper package takes care of all book-keeping functionality with respect to handling links, time-related conversion and spatial transformations (by use of the element mapper described above). When using the wrapper packages for model migration, implementation must follow the design pattern shown in Figure 10. Model migration will typically follow the steps described below:

1. The engine core must be turned into a component, so that it can be accessed from outside. If the engine is programmed in Fortran, for example, and is compiled into an executable file (EXE), this engine could be changed so that it can be compiled into a DLL, which can be accessed from outside through the Win32 API (MyEngineDll in Figure 10).
2. The engine must be changed so that initialization, performing a timestep and finalization are separate functions that can be accessed from outside. Initialization includes reading input files (populating the model), memory allocation and whatever model-specific actions need to be completed before calculations start. The function to perform a timestep triggers the model engine to make a single timestep. The finalization function takes care of engine-specific tasks that need to

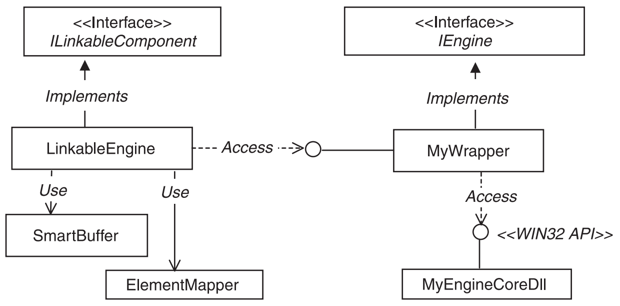


Figure 10 | Model engine wrapping design pattern.

be done after all calculations have been completed; this typically includes de-allocation of memory and closing output files.

3. The default implementation of the ILinkableComponent interface (LinkableEngine in Figure 10) in the wrapper packages is a generic implementation that suits most model engines of the time-stepping kind. Implementation tailored to the specific model that is being migrated is done in the MyEngineWrapper class (Figure 10). The MyEngineWrapper class must implement the IEngine interface (Figure 11). The LinkableEngine class accesses MyEngineWrapper through this interface. In practice, the implementation of MyEngineWrapper is done by creating a new class, subsequently using the development environment to auto-generate the stub code for the IEngine interface. When this has been done, the task is to fill in the functionality for each function. For some functions, such as the GetComponentID, the full implementation can take place in the MyEngineWrapper class; for other functions, interactions with MyEngineDll are needed. The latter typically involves extension of the API of MyEngineDll. It is usually advantageous to put the bulk of the implementation into the MyEngineWrapper class and only make small changes to the engine core. In this way most OpenMI-specific implementations can be done outside the engine core, which then remains intact to be used for other purposes.

For time-stepping model engines, the ILinkableEngine interface (Figure 11) is more straightforward to implement compared to the ILinkableComponent interface. For example, the GetValues method defined in the IEngine interfaces does not trigger any calculations; it simply returns the current values for internal variables identified by the method arguments 'QuantityID' and 'ElementSetID'. These identifiers can be recognized by the engine because

<<interface>> org.OpenMI.Utilities.IEngine

```
//== The org.OpenMI.Utilities.Wrapper.IEngine interface ==
// -- Execution control methods (Inherited from IRunEngine) --
void Initialize(Hashtable properties);
bool PerformTimeStep();
void Finish();
//-- Time methods (Inherited from IRunEngine) --
ITime GetCurrentTime();
ITime GetInputTime(string QuantityID,string ElementSetID);
ITimeStamp GetEarliestNeededTime();
//-- Data access methods (Inherited from IRunEngine) --
void SetValues(string QuantityID, string ElementSetID, IValueSet values);
IValueSet GetValues(string QuantityID, string ElementSetID);
//-- Component description methods (Inherited from IRunEngine) --
double GetMissingValueDefinition();
string GetComponentID();
string GetComponentDescription();
// -- Model description methods --
string GetModelID();
string GetModelDescription();
double GetTimeHorizon();
// -- Exchange items --
int GetInputExchangeItemCount();
int GetOutputExchangeItemCount();
org.OpenMI.Backbone GetInputExchangeItem(int exchangeItemIndex);
org.OpenMI.Backbone GetOutputExchangeItem(int exchangeItemIndex);
```

Figure 11 | Model engine interface.

they are the same as those exposed by the engine though the GetOutputExchangeItems method in the ILinkableEngine interface.

It is important to note that the ILinkableEngine interface is not part of the OpenMI standard. This interface is used only if the wrapper package is chosen for the model migration. This demonstrates how the standard tackles the potentially conflicting demands of giving the model provider the freedom to make the optimal implementation for a specific model while, at the same time, making it easy to migrate a model. By using the wrapper package, model migration can be done quite fast but, since the implementation of the wrapper package is done generically, this solution may not be optimal for every model engine. We envisage that people will generally use the wrapper package to the full extent the first time a model is migrated but, later on, gradually change the implementations to be more optimal for specific model engines.

A detailed description of the wrapper package is given in Sinding *et al.* (2005) and guidelines for model migration are given in the OpenMI Guidelines (Tindall *et al.* 2005).

CONCLUSIONS

The OpenMI prescribes a standardized way to access model components. The OpenMI standard is defined by the use of component interfaces. The OpenMI is intended to provide opportunities rather than being a straitjacket. Consequently the OpenMI interfaces have been made as simple as possible, in order to leave as much room as possible for the model provider to make the best solutions. We realize that such a high level of freedom may also make the standard difficult to use. Therefore, much effort has been put into the associated documentation, guidelines, design pattern and assisting software utilities and tools. In this way the normally contradictory demands for flexibility and ease of use have been tackled. Users can apply the standard alone, with its high level of flexibility, or base their work on the predefined design pattern and available software utilities in order to make the task easier.

Standards are a means for enabling people to work together. Standards like XML and HTTP are excellent examples of this, with developers around the world contributing tools and applications that use these standards and a wide range of software packages becoming available for end users. A standard will only become successful if it is technically sound, if the number of people using the standard reaches a critical level, and if the standard is supported and continuously developed in order to meet the demands that arise.

In order to make sure that the OpenMI is technically sound, a large number of user cases have been tested, where very complex model systems have been linked using the OpenMI. The ease of use and usefulness of the OpenMI have been tested in the proof-of-concept phase of the HarmonIT project, by having eight different universities and companies migrating and linking models using the software tools and guidelines provided. The encouraging results from this exercise were presented at the final HarmonIT workshop in Munich, where feedback for further improvements was also provided.

OpenMI makes it possible to create software systems for integrated catchment management by means of linking available OpenMI-compliant models. These models may come from different providers and the system builder can choose whatever model is most suited for each specific case, which means that high quality systems that represent the underlying physics accurately can be developed. Naturally, this is only possible if there is a wide variety of OpenMI-compliant models available. Therefore one of the main tasks for the HarmonIT project was to migrate about 20 different models, including some of the most widely used commercial hydrological and hydraulic models. After the release of OpenMI version 1.0, people outside the development group have also started to migrate models (e.g. Visual MODFLOW). The more available OpenMI-compliant models there are, the more attractive it becomes to use OpenMI, and we hope that this means that the number of available OpenMI-compliant models will keep growing.

It is important for the survival of OpenMI that it is being supported and continuously developed to meet new demands. The challenge is to ensure a balance between keeping the standard static in order not to burden people with demands for upgrading their components, yet still develop and react to change requests. This will be the responsibility of the OpenMI association that is currently being established. The OpenMI association will be an open forum and we hope that people around the world will join and contribute to the future of OpenMI.

OpenMI was developed with the aim of enabling the dynamic linking of hydraulic and hydrological models. However, it turned out that OpenMI can do much more than that. The OpenMI can be used for any component that can accept or provide data, which means that databases, flat data files or even on-line data can be turned into LinkableComponents and become an integrated part of OpenMI configurations. Since the OpenMI provides a standardized way to interact with model components, we also see opportunities to develop tools for auto-calibration, optimization and scenario management. Such tools can, once developed, be seamlessly applied with any OpenMI-compliant model. With such tools available, even more powerful modelling systems can be created for the benefit of end users and water management decision-makers.

REFERENCES

Abbott, M. B., Bathurst, J. C., Cunge, J. A., O'Connell, P. E. & Rasmussen, J. 1986 An introduction to the European Hydrological System-Systeme Hydrologique Euopeen, 'SHE', 2: structure of a physically-based, distributed modelling system. *J. Hydrol.* **87**, 61–77.

Gijsbers P., Brinkman R., Gregersen J., Hummel, S. & Westen S. 2005 *The OpenMI Document Series: Part C The org.OpenMI.Standard Interface Specification (version 1.0)*. Available at: http://www.OpenMI.org.

Moore R., Tindall, I., Gijsbers, P., Fortune, D., Gregersen, J. & Blind, M. 2005 *OpenMI Document Series: Part A Scope for the OpenMI (version 1.0)*. Available at: http://www.OpenMI.org.

Sinding, P., Gregersen, J., Gijsbers, P., Brinkman, R. & Westen, S. 2005 *The OpenMI Document Series: Part F org.OpenMI.Utilities Technical Documentation (version 1.0)*. Available at: http://www.OpenMI.org.

SourceFORGE 2005 *OpenMI Open Source*. Available at: http://sourceforge.net/projects/openmi.

Tindall, I., Gijsbers, P., Gregersen, J., Westen, S., Dirksen, F., Gavardinas, C. & Blind, M. 2005 *OpenMI Document Series: Part B Guidelines for the OpenMI (version 1.0)*. Available at: http://www.OpenMI.org.